

University of Alberta

**Synthesis of Adaptive Hardware System
on Field Programmable Gate Arrays**

by

Gautam Subhash Karnik



A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the
requirements for the degree of Master of Science

in

Department of Electrical and Computer Engineering

Edmonton, Alberta

Fall 2003



National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file *Votre référence*

ISBN: 0-612-87796-5

Our file *Notre référence*

ISBN: 0-612-87796-5

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

Canada

University of Alberta

Library Release Form

Name of Author: Gautam Subhash Karnik

Title of Thesis: Synthesis of Adaptive Hardware System on Field Programmable Gate Arrays

Degree: Master of Science

Year this Degree Granted: 2003

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Date: May 14, 2003

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled Synthesis of Evolvable Hardware System on Field Programmable Gate Arrays submitted by Gautam Subhash Karnik in partial fulfillment of the requirements for the degree of Master of Science

Dr. Marek Reformat (Co-supervisor)

Dr. Witold Pedrycz (Co-supervisor)

Dr. Xiaoling Sun (Internal Examiner)

Dr. Eleni Stroulia (External Examiner)

Date: April 30, 2003

Abstract

Recently, research in Evolutionary Computation has switched focus to applications in Electrical Engineering, leading to the field of study called Evolvable Hardware. The ultimate goal is to develop hardware systems with on-line adaptation capabilities. Applications requiring optimum performance due to changing environments can be integrated with an evolutionary component capable of fine-tuning system parameters during real-time operation. The proposed Adaptive Hardware System consists of an Evolutionary Adaptation System, Configurable Working System, and On-line Monitoring and Diagnostic System. Focus is primarily concentrated on the Evolutionary Adaptation System called the Autonomous Genetic Machine. Synthesis results for Field Programmable Gate Arrays from various manufacturers are studied. Performance measurements relating chromosome length, population size, fitness representation and system clock frequency are investigated. As an extension, the Fitness Evaluation Subsystem is developed. Capable of configuring and evaluating an application using chromosomes in conjunction with training data, this additional subsystem increases the scope of potential applications.

Dedications and Acknowledgements

I would like to dedicate this thesis to my parents, Subhash and Medha Karnik, my brother, Nachiket Karnik and other family members.

I would like to acknowledge the support given by my supervisors, Dr. Marek Reformat and Dr. Witold Pedrycz, during my graduate studies at University of Alberta.

Table of Contents

CHAPTER ONE: INTRODUCTION TO EVOLVABLE HARDWARE.....	1
1.1 PARAMETRIC DESIGN AND DESCRIPTIONS OF ELECTRONIC HARDWARE.....	1
1.2 PROGRAMMABLE HARDWARE – INTRINSIC AND EXTRINSIC APPROACHES	3
1.3 HARDWARE IMPLEMENTATIONS OF GENETIC ALGORITHMS.....	5
1.4 EVOLVABLE SYSTEM-ON-A-CHIP	6
CHAPTER TWO: INTRODUCTION TO GENETIC ALGORITHMS	8
2.1 SELECTION METHODS.....	9
2.2 GENETIC OPERATIONS	10
2.3 FITNESS REPRESENTATION.....	12
2.4 STEADY STATE GENETIC ALGORITHM.....	13
CHAPTER THREE: ADAPTIVE HARDWARE SYSTEM	15
3.1 ADAPTIVE HARDWARE APPROACHES	15
3.2 REALIZATION OF EVOLUTIONARY-BASED ADAPTATION SYSTEM	18
CHAPTER FOUR: AUTONOMOUS GENETIC MACHINE.....	20
4.1 CONCEPT OF AUTONOMOUS GENETIC MACHINE	20
4.2 DESIGN OF AUTONOMOUS GENETIC MACHINE.....	21
4.2.1 <i>High-Level Architecture</i>	23
4.2.2 <i>Controller</i>	24
4.2.3 <i>Memory Bus Controller</i>	26
4.2.4 <i>Initialization</i>	26
4.2.5 <i>Evaluation</i>	26
4.2.6 <i>Selection</i>	28
4.2.7 <i>Reproduction</i>	28
4.2.8 <i>Population Replacement</i>	29
4.2.9 <i>Pseudo-Random Number Generation</i>	29
4.3 HARDWARE IMPLEMENTATION OF EVOLUTIONARY ENVIRONMENT	29
4.3.1 <i>Memory</i>	29
4.3.2 <i>Controller</i>	30
4.3.3 <i>Initialization</i>	31
4.3.4 <i>Evaluation Controller</i>	32
4.3.5 <i>Selection</i>	33
4.3.6 <i>Reproduction</i>	34
4.3.7 <i>Population Replacement</i>	35
4.3.8 <i>Pseudo-Random Number Generation</i>	35
4.4 EXPERIMENTS WITH NUMERICAL OPTIMIZATION	36
4.5 ANALYSIS OF AUTONOMOUS GENETIC MACHINE.....	42
4.5.1 <i>Population Size and Chromosome Length</i>	43
4.5.2 <i>Synthesis of AGM</i>	46
4.5.3 <i>Effects of Fitness Representation</i>	52
4.6 BOTTLENECKS.....	53
4.6.1 <i>Termination Criteria</i>	53

4.6.2 Tournament Selection	54
4.6.3 Reproduction	54
4.6.4 Population Replacement	54
4.7 CONCLUSIONS	55
CHAPTER FIVE: FITNESS EVALUATION SUBSYSTEM	56
5.1 ADAPTIVE HARDWARE SYSTEM OVERVIEW	56
5.2 DESIGN OF FITNESS EVALUATION SUBSYSTEM	57
5.2.1 High Level System Architecture	58
5.2.2 Training Controller	60
5.2.3 Input Preparation Unit	61
5.2.4 Output Processing Unit	61
5.2.5 Application-Dependant System	61
5.3 LINEAR REGRESSION	62
5.4 ANALYSIS OF FITNESS EVALUATION SUBSYSTEM	64
5.4.1 Synthesis of FES	64
5.4.2 Effects of Fitness Representation	65
5.5 CONCLUSIONS	66
CHAPTER SIX: HARDWARE IMPLEMENTATION	67
6.1 HIGH LEVEL DESIGN	67
6.2 EXECUTION STEPS AND PIN ASSIGNMENTS	68
6.3 EXPERIMENTAL RESULTS USING LINEAR REGRESSION	69
6.4 CONCLUSIONS	70
CHAPTER SEVEN: CONCLUSIONS AND FUTURE WORK	71
7.1 CONCLUSIONS	71
7.2 FUTURE WORK	72
BIBLIOGRAPHY	75

List of Figures

FIGURE 1.1A. EVOLUTIONARY SYNTHESIS OF ELECTRONIC CIRCUITS	4
FIGURE 1.1B. SYSTEM-ON-A-CHIP EVOLUTIONARY DESIGN.....	4
FIGURE 2.1 CROSSOVER OPERATIONS	10
FIGURE 2.2 MUTATION OPERATIONS	11
FIGURE 3.1 ON-LINE ADAPTIVE HARDWARE SYSTEM.....	17
FIGURE 3.2 CONCEPT OF AUTONOMOUS GENETIC MACHINE	18
FIGURE 3.3 SYSTEM CONFIGURATIONS FOR EXTERNAL EVALUATION COMPONENT	19
FIGURE 4.1 CONCEPT OF AUTONOMOUS GENETIC MACHINE	21
FIGURE 4.2 ARCHITECTURE FOR AUTONOMOUS GENETIC MACHINE.....	22
FIGURE 4.3 ARCHITECTURE FOR EVALUATION CONTROLLER.....	27
FIGURE 4.4 INTERFACE FOR MEMORY COMPONENT	29
FIGURE 4.5 INTERFACE FOR CONTROLLER	30
FIGURE 4.6 INTERFACE FOR INITIALIZATION COMPONENT.....	31
FIGURE 4.7 INTERFACES FOR INTERNAL COMPONENTS IN EVALUATION CONTROLLER.....	32
FIGURE 4.8 INTERFACES FOR INTERNAL COMPONENTS IN SELECTION COMPONENT.....	33
FIGURE 4.9 INTERFACES FOR INTERNAL COMPONENTS IN REPRODUCTION COMPONENT	34
FIGURE 4.10 INTERFACE FOR REPLACEMENT COMPONENT	35
FIGURE 4.11 INTERFACE FOR LFSR.....	35
FIGURE 4.12 POLYNOMIAL FUNCTIONS INTEGRATED WITH AGM.....	36
FIGURE 4.13A. FITNESS LANDSCAPE FOR FIRST 25 GENERATIONS.....	38
FIGURE 4.13B. FITNESS LANDSCAPE FOR GENERATIONS 25 TO 60.....	38
FIGURE 4.13C. FITNESS LANDSCAPE FOR GENERATIONS 60 TO 100.....	39
FIGURE 4.14A. EFFECTS OF CHROMOSOME LENGTH ON POPULATION FOR FLEX 10 KE FPGA	44
FIGURE 4.14B. EFFECTS OF CHROMOSOME LENGTH ON POPULATION FOR VIRTEX 1000 FPGA	45
FIGURE 4.15 EFFECTS OF POPULATION SIZE ON SYSTEM PERFORMANCE FOR FLEX 10KE AND VIRTEX 1000 FPGAS.....	46

FIGURE 4.16 EFFECTS OF FITNESS ON SYSTEM CLOCK FREQUENCY	53
FIGURE 5.1 CONCEPT OF ADAPTIVE HARDWARE SYSTEM	57
FIGURE 5.2 ARCHITECTURE OF FITNESS EVALUATION SUBSYSTEM.....	59
FIGURE 5.3 COMPARISON OF TARGET DATA WITH OPTIMIZED RESULTS FROM AGM.....	63
FIGURE 5.4 EFFECTS OF FITNESS REPRESENTATION ON MAXIMUM CLOCK FREQUENCY FOR FLEX 10 KE AND VIRTEX FPGA	65
FIGURE 6.1 TOP LEVEL OF DESIGN ON FPGA	68

List of Tables

TABLE 4.1 AGM RESULTS FOR POLYNOMIAL FUNCTION 1	37
TABLE 4.2 AGM RESULTS FOR POLYNOMIAL FUNCTION 1	39
TABLE 4.3 AGM RESULTS FOR POLYNOMIAL FUNCTION 2	41
TABLE 4.4 AGM RESULTS FOR MAXIMUM OF POLYNOMIAL FUNCTION 3.....	42
TABLE 4.5 AGM RESULTS FOR MINIMUM OF POLYNOMIAL FUNCTION 3.....	42
TABLE 4.6 AGM COMPONENT CHARACTERISTICS FOR FLEX 10 KE	46
TABLE 4.7 AGM COMPONENT CHARACTERISTICS FOR VIRTEX 1000	47
TABLE 4.8 AGM COMPONENT CHARACTERISTICS FOR VARIOUS FPGAS	48
TABLE 4.9 PERFORMANCE OF AGM	51
TABLE 5.1 EXPERIMENTAL RESULTS FOR LINEAR REGRESSION TESTS	62
TABLE 5.4 LOGIC CELL USE AND CLOCK FREQUENCY OF FES ON FLEX 10KE FPGA.....	64
TABLE 5.5 LOGIC CELL USE AND CLOCK FREQUENCY OF FES ON VIRTEX 1000 FPGA.....	64
TABLE 6.1 PIN ASSIGNMENTS OF ADAPTIVE HARDWARE SYSTEM ON XSV BOARD	69
TABLE 6.2 SIMULATIONS RESULTS FOR LINEAR REGRESSION TESTS	70
TABLE 6.3 IMPLEMENTATION RESULTS FOR LINEAR REGRESSION TESTS	70

Chapter One: Introduction to Evolvable Hardware

Research in Evolutionary Computation has switched some of its focus to applications in Electrical Engineering problems. Scientists and engineers have begun to use Genetic Algorithm to aid in the design and optimization of electrical circuits, leading to the field of study called Evolvable Hardware (EHW). The most practical use of EHW is to fine tune parameters of existing hardware designs [25]. For example, optimization of circuit parameters at the transistor level can be done in order to increase performance with respect to circuit board defects and environmental conditions. Researchers [13][4] have extended EHW to the design of hardware systems at the gate level given a set of specifications. Still, others [12] have applied EHW to Test Automation and Circuit Layout problems. EHW has been used to reconfigure connections on electronic hardware such as Field Programmable Gate Arrays (FPGA) [12][19]. In the most abstract sense, EHW refers to the use of Evolutionary/Genetic Algorithms to aid in any aspect of the hardware design process. The final goal is the creation of complete evolvable hardware systems that can adapt to changing environments and increase system performance during normal operation [17].

Evolvable hardware can be summarized five main categories, 1) Evolutionary Search for Parametric Design, 2) Evolution of Descriptions of Electronic Hardware, 3) Evolution of Programmable Hardware Downloaded through Software, 4) Evolvable System-On-A-Chip (SOC), and 5) Evolvable Systems integrated into standard systems adding an "evolvable" feature [17].

1.1 Parametric Design and Descriptions of Electronic Hardware

Zebulum, Pacheco and Vellasco [25] used Genetic Algorithms to aid in the synthesis of CMOS Operational Amplifiers. The topology of the circuit was known in advance; the Genetic Algorithm was used to manipulate the transistor dimensions, biasing current and compensating capacitor in order to optimize the gain, power consumption, area, phase margin and slew rate. The evaluation of each individual was done with the SMASH simulator. Fitness calculations were based on a multi-objective evaluation function by aggregating all objectives together using a weighted sum. In this study, the Genetic Algorithm produced results competitive with human designed cells based on standard design techniques without any previous knowledge being supplied.

Miller, Job and Vassilev [13] investigated the principles in evolutionary design of digital circuits. Symbols representing logic level 0 and 1, primary inputs and output and various combinations of logic gates such as AND, OR, XOR, MUX, and INVERT were the basic building blocks used in the chromosome. Efforts were made to find an evolved version of a one-bit carry adder, two-bit carry adder, two-bit multiplier, three-bit multiplier and even four-bit parity checker that used less

gate than the conventional circuits used in industry standard macro cell libraries. In all cases, more effective circuits were discovered however, the computational effort increased significantly as the size of each problem increased. For the one-bit adder with carry, approximately 10,000 generations was needed before a more effective design was discovered. When the two-bit adder was evolved, 50,000 generations was adequate. The two-bit multiplier required 100,000 generations and the three-bit multiplier required 4 million generations. The four-bit parity checker was built using AND, NAND, OR and NOR gates, and required 1 million generations before perfect solutions were found. Due to rigid nature of the digital domain, Evolutionary principles must find perfect solutions as opposed to finding optimal solutions for which Evolutionary/Genetic Algorithms are suited. In order to minimize the effect of this growing complexity, the fitness landscapes in each experiment need to be examined more closely in order to develop an evolutionary approach that can reduce the computational effort required. Perhaps use of fitness functions that are more relaxed initially and increase in rigidity as the algorithm progresses could prove to be an effective method of reducing computational effort.

Vassilev and Miller [24] continued work to find a way of integrating groups of logic gates or building blocks that could aid in yielding desired results faster. They analyzed their results from the three-bit multiplier and discovered repeated combinations of XOR and AND gates. They created nine more subcomponents with these building blocks and repeated their experiment. Results were still inconclusive. One run took approximately 400,000 generations while another took 5.6 million generations. When the final circuits were analyzed, it was shown that the Evolutionary Algorithm rebuilt the building blocks on its own despite the fact that the other components were available. Whether the method of identifying building blocks is manual or automatic, the key to integrating building blocks in to this process is to define building blocks that are suitable for evolution.

Coello, Christiansen, Aguirre [4] attempted a similar problem and obtained completely different results. In their experiments they limited the maximum size of the circuit and used only AND, OR, XOR and INVERT gates. Their chromosome structure was made simpler too in that gates were only allowed to communicate with outputs from the previous layer in the two-dimensional template. A three-bit input, one-bit output problem was solved in 81 generations with population size 300. Another four-bit input, one-bit output problem was solved in 99 generations with population size 1000. In this case, their Genetic Algorithm was run for 300 generations in order to optimize the number of gates required to solve the problem. The Genetic Algorithm solved the problem with 4 gates, whereas two human designers solved the problem with 5 and 6 gates respectively. In their third experiment, another four-bit input problem was examined and the Genetic Algorithm came up with a solution in 155 generations with a population size of 2000. The

GA solved the problem with 7 gates and the two human designers took 9 and 10 gates respectively. The next set of tests involved multiple input and multiple output tests. A two-bit multiplier circuit was synthesized with this method. Convergence to the optimal solution was achieved in 220 generations and required only 7 gates compared to research from Miller et al., which required 9 gates after 100,000 generations. Human designers used 8 and 12 gates. In the final experiment, a four-bit input, three-bit output experiment was examined. Using a population size of 2800 yielded an optimal solution after only 1059 generations.

The major differences between Miller et al. and Coello et al. occur at a couple of levels. First, Miller used Evolutionary Algorithms that only employ mutation operations while Coello used Genetic Algorithm that combine the crossover and mutation operations together. This way, genetic traits from the most successful individuals are passed on to the next generation. Second, Miller et al. used a chromosome that gave the Evolutionary Algorithm much more flexibility in the topology of the circuit while Coello et al., used a fixed topology with less flexibility. This constraint on the chromosomes most likely aided the Genetic Algorithm because the size of its search space was significantly reduced. By giving the Evolutionary/Genetic Algorithm too much flexibility, the size of the search space increases dramatically requiring more computational effort. More work into assessing the degree of human intervention, identification of building blocks and proper analysis of fitness landscapes are needed in order to draw practical conclusions about research at this level, nevertheless, the results are promising.

1.2 Programmable Hardware – Intrinsic and Extrinsic Approaches

Stoica and Thompson [17][19] attempted to define the main steps involved in the evolutionary design of electric circuits. Two strategies were devised, extrinsic and intrinsic evolution. A solution devised under extrinsic solution is designed as a blueprint for hardware that may be eventually downloaded into some kind of configurable device. Evaluation of each chromosome is done on a computer using a circuit simulator like SPICE or SMASH. Intrinsic evolution involves converting chromosomes into a configuration bit string that is downloaded into a programmable device. Circuit responses are compared against target responses in order to ascertain and rank each individual. Figure 1.1a [17] illustrates this concept.

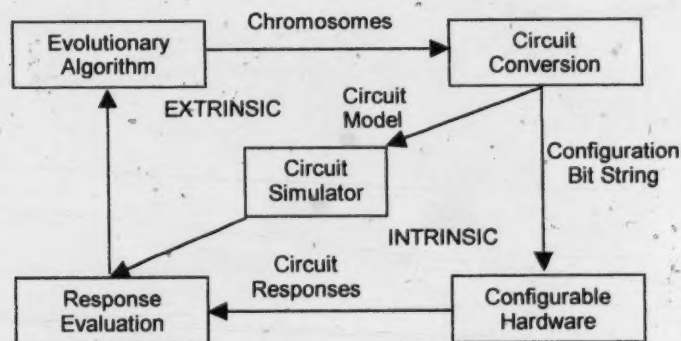


Figure 1.1a. Evolutionary synthesis of electronic circuits

Stoica used this model to evolve analogue circuits extrinsically in PSPICE as well as intrinsically using Field Programmable Transistor Arrays (FPTAs). The goal of this work was to approximate fuzzy T-norms and T-conorms in hardware for use in fuzzy neural networks. In most experiments, the target response was achieved for the fuzzy operations. Not only were the results promising, but the results were obtained in 400 generations using only a population size of 128 individuals. The next phase of this work involved the extension of evolutionary processes to fault tolerant systems. In these set of experiments, faults were injected into the FPTAs in order to see if the intrinsic system could adapt. Results indicated that 90% of circuit performance could be recovered. Stoica proposed the next stage of this work could involve the integration of an evolutionary environment to a system-on-a-chip. In such a system, the evolutionary unit would be used inside an FPGA in order to reconfigure another part of the chip, Figure 1b. The chromosomes would map directly to a configuration for a digital system. System responses could be collected and characterized as a fitness value for the Evolutionary Algorithm.

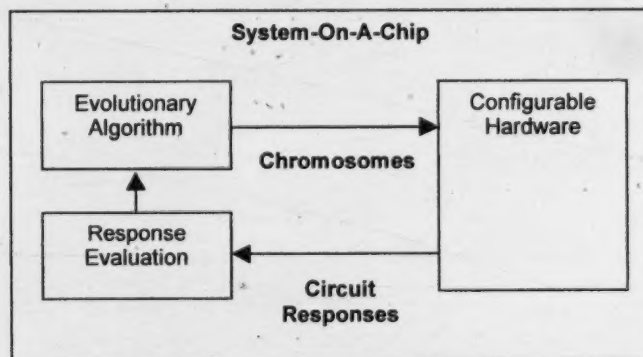


Figure 1.1b. System-On-A-Chip Evolutionary Design

Extensions of this work could involve the creation of evolvable space systems that have the ability to adapt to new functionality due to changes in environment or new missions, thereby increasing the lifespan of spacecraft systems. Extensions include on-board sensors, antennas, mechanical and optical subsystems.

Thompson et al., [19] used a personal computer to configure the connections between logic cells in a 10x10 corner of the XC6216 Field Programmable Gate Array (FPGA) using an evolutionary approach. The objective was to create an evolved circuit that could discriminate between square waves of 1 kHz and 10 kHz. The experiment took three weeks and the result was an evolved circuit that was small and able to behave correctly in $\pm 5^\circ\text{C}$ range but the circuit did not have clear functional decomposition and could only run on the particular FPGA for which it was evolved. However, this work clearly demonstrates the feasibility of using an evolutionary process to re-configure hardware.

Thompson [18][19] investigated the use of the evolutionary process in the production of fault tolerant designs. A Dynamic State Machine (DSM) was designed to control movements of a robot in order to always keep away from enclosed walls using motors and sonar devices. The evolutionary process had control of a Genetic Latch, that could be manipulated to let some signals pass through asynchronously and other signals pass synchronously. The Genetic Latch had control over which inputs and outputs were clocked. Also, the RAM components for the state machine used by the Genetic Latch was under control of the genetic algorithm. Experiments were conducted where single-stuck-at faults were inserted in the RAM components containing next state information. It was found that faults had the same effect on the behavior of the robot, as genetic mutations. That is, evolved systems were less sensitive to faults than equivalent systems produced by standard means and evolution continued to take place as if the faults were a part of the original system. However, it was not empirically shown how much of this effect was attributed to the design of the DSM or the evolutionary process. Nevertheless, this work shows that evolutionary process can be applied to preexisting hardware designs that are robust enough to be reconfigured by Genetic Algorithms.

1.3 Hardware Implementations of Genetic Algorithms

Other attempts have been made to design an evolutionary environment for synthesis onto programmable logic. Tommisksa and Vuori, [20] designed a Genetic Algorithm for Altera's Flex 10K FPGA Family. Their implementation was divided into four main pipeline stages and was designed with performance issues in mind. Their system implemented a Steady State Genetic Algorithm where only two parents are selected and replaced by their offspring. This process ensures that there are no copies of the same individuals in the population. The pipeline approach

will definitely yield a significant gain in performance over a conventional software approach. However, it can take a long time to converge to an optimal solution. In Steady State Genetic Algorithms only two parents are selected for reproduction in each generation, hence, the population does not vary greatly from generation to generation. Also, their system integrated the evaluation of individuals with the evolutionary process. If changes to fitness calculations are ever required, then a new pipeline must be constructed and synthesized.

Scott, Seth and Samal, [15] attempted to create a hardware-based Genetic Algorithm where a personal computer communicated with an evolutionary process running on programmable application specific hardware. In their system, the personal computer interacted with the evolutionary process via shared memory. Simulations were done using VHDL (Very High Speed ASIC Hardware Description Language) in order to assess the design. They created their evolutionary system generic enough to be integrated with more than one application. However, changes to the fitness calculations require the reconstruction, recompilation and verification of the hardware system. The aim of this hardware implementation was to reduce memory usage during operation of the Genetic Algorithm. It was achieved by swapping the main population and intermediate population memory for every generation, however, in such a design there was no definitive modular block in the system containing the final population, making it hard to monitor the memory usage of the Genetic Algorithm during execution. Nevertheless, both approaches show the feasibility of extending the intrinsic and extrinsic approaches to the System-On-A-Chip approach.

1.4 Evolvable System-On-A-Chip

Higuchi et al., [7][8][9] attempted to create an Evolvable Hardware system completely designed onto a single chip. Genetic Algorithm operations were used for the on-line adaptation of re-configurable Programmable Logic Arrays (PLA). Inside the GA Unit on the chip, two PLAs were used to evaluate the offspring produced from crossover and mutation; the actual PLA being used in the system was external to GA Unit. Simulations were conducted on an application of this chip to the control of an artificial hand. Learning through neural networks needed around three hours to train the circuits but in simulations the EHW chip would take 80 milliseconds. Further work was done where an EHW chip was used to re-configure clock timing distribution of flip-flop registers inside a digital system. It was shown in simulations that 50% of the experimental circuits designed for 500 MHz could be adjusted to function at 800 MHz whereas without using the evolutionary approach only 2.9% of the chips could be adjusted. Though these approaches were novel, the design of these EHW chips was restricted to the specific applications they were designed for.

The System-On-A-Chip Evolvable Hardware approach presented in this study will contain a static portion of the FPGA consisting of a generic Genetic Algorithm hardware component, called the Autonomous Genetic Machine (AGM), made such that it is independent of the optimization problem being solved. The dynamic portion of the FPGA consists of a configurable digital system that could be manipulated directly with a binary encoded chromosome. Such an Adaptive Hardware System (AHS) could be integrated into a digital system where there are two modes of operation, normal operation and adaptive operation. Digital systems could then be re-configured or adapted on-line as environmental conditions or design specifications change.

Chapter Two: Introduction to Genetic Algorithms

Evolutionary Computation is an optimization technique that attempts to mimic the process of biological evolution in order to find the optimal solution for a particular application. Due to the abstract nature of this approach, there is a broad range of problems that can be used with Evolutionary Computation techniques. Image classification, signal processing, intrusion detection, biotechnology, spacecraft attitude maneuvers, financial market predication, neural network training, database querying optimization, and computer graphics are just a few of the areas in which Evolutionary approaches have been applied [2].

Holland and Goldberg [5][10] developed the first evolutionary approach in the 1970s called Genetic Algorithms. In nature, individuals compete for resources in order to survive. The fittest individuals survive and pass on their genetic information to their offspring in order to propagate traits that can ensure the survival of the species. Sometimes mutations can occur that either degrade the performance of individuals or give individuals an even more competitive edge. Eventually, the population inherits positive changes caused by mutations. Just as guanine, adenine, cytosine and thiamine make up the genetic code of humans, problems can be broken down into functional components and represented in a similar manner too. In Genetic Algorithms, potential solutions or designs for a problem are encoded as binary valued strings where one bit is analogous to a gene and the entire string is analogous to a chromosome. Each chromosome represents a particular solution to the problem being investigated. A set or population of these candidate solutions is generated and evaluated. The chromosomes or individuals that come closest to yielding a desired solution are allowed to reproduce and pass on their genetic makeup to the next generation. Mutations are allowed to occur during the reproduction in order to ensure variability in the population. These steps are repeated for a maximum number of cycles, called generations. Eventually, this process will come up with a pool of solutions either close to or matching the optimal results expected for the problem being investigated. Interestingly, some applications using Genetic Algorithms have realized solutions that have already been patented.

Other paradigms in the field of evolutionary computation include Evolutionary Algorithms, Genetic Programming [11], Evolutionary Strategies and Learning Classifier Systems. Evolutionary Algorithms use only mutations to evolve populations. Genetic Programming extends the concept of Genetic Algorithms to the generation of computer programs using LISP programming expressions called S-lists. The chromosome consists of a series of programming instructions instead of binary-valued parameters. Likewise, Evolutionary Strategies incorporate some features of Genetic Algorithms but use real-valued parameters in place of binary-valued

parameters. Another extension of Genetic Algorithms called Learning Classifier Systems, attempts to evolve populations of condition/action rules [27].

The stages in the Genetic Algorithm are abstract enough to be applied to quite a variety of problems. The main issue lies in creating an adequate chromosome to represent then problem and evaluating the fitness of each individual. The rest of the Genetic process is left untouched.

The Canonical Genetic Algorithm can be expressed as follows [12]:

1. Generate initial population,
2. Evaluate each individual and assign fitness value,
3. Select fittest individuals such that the probability of selection of each individual is proportional to its fitness,
4. Pair the individuals randomly to form parents,
5. With a probability, P_c , perform **crossover** on the parents to generate two offspring otherwise clone the parents ($P_c \cong 0.6 - 0.9$),
6. **Mutate** the offspring with a probability P_m otherwise leave the offspring untouched ($P_m \cong 0.001 - 0.3$),
7. Perform **inversion** (mutation on a series of bits) on the offspring with probability P_i if the application requires it,
8. Replace all the individuals of the previous generation with the parents, clones and offspring,
9. Check **termination criteria**. If criteria not met, go back to Step 2. Else go to Step 10,
10. Display results.

2.1 Selection Methods

Each stage of the Genetic Algorithm can be implemented using a variety of different methods. For example, selection criteria can simply be implemented as a tournament between two randomly selected individuals, or using some probabilistic methods such as Roulette Wheel Selection or Stochastic Universal Selection. In binary tournament selection, two candidates are taken from the population at random and the better individual is selected as a parent. This process is repeated again to obtain the next parent. This method ensures that all individuals in the population have the chance to take part in reproduction. Sometimes parents with a poor fitness can produce an extremely good offspring if crossed with a strong parent, or a simple mutation on a poor parent can produce fit offspring. In some approaches, the selected parents are taken out of the main population such that the offspring only replace the parents. In other approaches the parents are placed in an intermediate population that also contains their offspring. This intermediate population is used to replace the previous generation. In Roulette Wheel selection, slots of a roulette wheel are sized according to the fitness of each individual in

the population. An individual is selected as a parent by spinning the roulette wheel and noting the position of the marker. Therefore, the probability of selecting an individual is proportional to its fitness. In Stochastic Universal Selection, N equidistant markers are placed around the wheel, where N is the population size. N individuals are selected in a single spin of the wheel, and the number of individuals selected is equal to then number of markers. Regardless of the type of selection method used, the overall flow of the algorithm is not disrupted though the final results may differ. *The hardware based Genetic Algorithm presented in this study, implements selection as binary tournament selection since it is the simplest approach, using the least amount of hardware overhead and yielding the fastest execution time.*

2.2 Genetic Operations

Crossover can be implemented as one or two point crossover, where one or two positions are randomly selected in the chromosome and genetic information is exchanged within these points between the parents. Uniform crossover occurs when each chromosome position is exchanged with some probability, usually one-half, between the parents. Mutation can be implemented as the inversion of a single bit in the offspring or a series of bits. Again, the overall flow of the algorithm is not disrupted however the final population will differ. The choice over the implementation of crossover and mutation depends largely on the amount of diversity one wants in the population. Figure 2.1 and Figure 2.2 illustrates the differences between the crossover and mutation operations. The bold characters mark the crossover and mutation points.

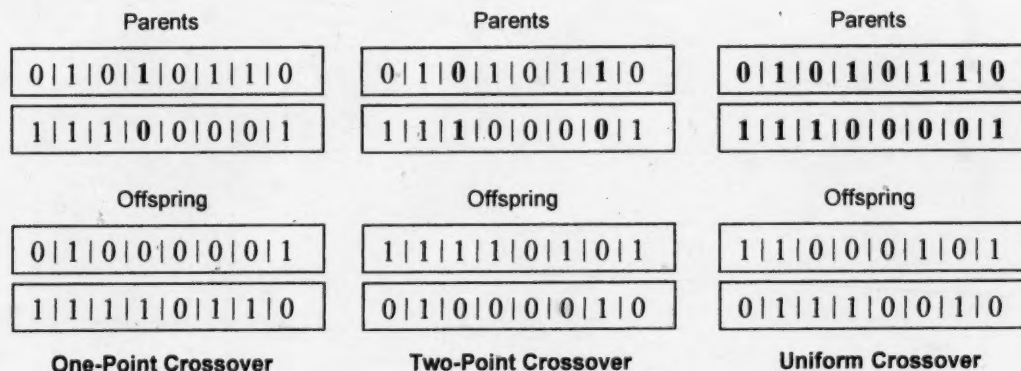


Figure 2.1 Crossover Operations

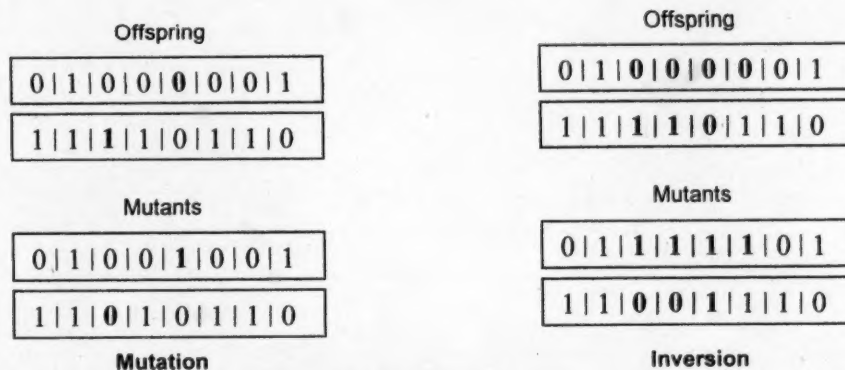


Figure 2.2 Mutation Operations

Reproduction is also quite an important process in the evolutionary process. It introduces genetic variations to the population. If these variations are too dramatic, the evolutionary algorithm will be analogous to a random search through the entire search space. If these variations are too small, then the evolutionary algorithm will focus in a region of local optima. There are three main kinds of strategies that have been developed for reproduction in evolutionary computation, being the $(1+1)$, $(\mu+1)$, (μ, λ) and $(\mu+\lambda)$ strategies [1][14]. In a $(1+1)$ strategy, the evolutionary process uses a population of one individual that generates one offspring by applying normally distributed mutations. If the child performs better than its parent, it replaces the parent. In the $(\mu+1)$ strategy, parents are allowed to form one offspring by recombination and mutation. The offspring will eventually replace the worst parent individuals if it is better than them. The (μ, λ) notation indicates that μ parents will create λ offspring by recombination and mutation, and the best offspring are deterministically selected to replace the parents. The $(\mu+\lambda)$ approach replaces the parents with the best offspring such that there is a guaranteed course of evolution. Generally, the (μ, λ) strategy is used due to its simplicity. In this work the (μ, λ) strategy is used to implement the hardware based Genetic Algorithm.

Some Genetic Algorithms integrate a technique called elitism, where the best individuals are passed on from one generation to the next. Elitism ensures that the fitness of the best individuals in the population does not decrease. Much research has gone into improving Genetic Algorithms; as the field matures, new approaches will give us more insight into how this process can be improved. The implementation of elitism in hardware would increase the computational time and add more hardware overhead. Such a feature, if added, would have to cycle through the entire population and save the best individuals in the intermediate population. Therefore, in the interest of reducing complexity of the hardware based Genetic Algorithm, elitism will not be used. It is assumed that the population will collectively evolve towards a viable solution, though the average fitness of the population will vary from one generation to the next.

2.3 Fitness Representation

Fitness is the driving force behind genetic algorithms because it characterizes the genotype of an individual as a phenotype. The method used in this process, can make a big difference in the selection of candidate parents during reproduction. Consequently, there are numerous methods of assessing the individuals in the population. However, there are four main methods that have been developed [11] being raw fitness, standardized fitness, adjusted fitness and normalized fitness.

Raw fitness is usually defined over a set of fitness cases that provide a basis for evaluating the fitness of an individual. Fitness cases are typically only a small finite sample of the entire domain of the problem being optimized. Raw fitness is usually described as the error, as is described as:

$$r(i,t) = \sum | S(i,j) - C(j) |,$$

where $r(i,t)$ is the raw fitness, $S(i,j)$ is the value returned by the individual, $C(j)$ is the correct value for the fitness case j .

Standardized fitness represents the raw fitness such that a lower numerical value is considered to be a better value. It is used for a particular problem when the lower value of raw fitness is preferred. For example, when trying to minimize a cost measure, the standardized fitness would be an appropriate measure to use. Standardized fitness is defined as:

$$s(i,t) = r_{\max} - r(i,t),$$

where $s(i,t)$ is the standardized fitness, $r_{\max}$ is the maximum possible value of raw fitness and $r(i,t)$ is the raw fitness.

Adjusted fitness is used when one wishes to exaggerate the small differences in fitness between individuals. The properties of this function, the adjusted fitness will make a clear distinction between a good individual and a very good individual. It is calculated from standardized fitness and is as follows:

$$a(i,t) = 1 / (1 + s(i,t)),$$

where $a(i,t)$ is the adjusted fitness, $s(i,t)$ is the standardized fitness for an individual i , at time t .

Normalized fitness provides a mechanism for evaluating individuals proportionally based on the combined fitness of the current population. Normalized fitness is expressed as:

$$n(i,t) = a(i,t) / \sum a(k,t),$$

where $n(i,t)$ is the normalized fitness, $a(i,t)$ is the adjusted fitness of an individual at time t and $\sum a(k,t)$ is the combined fitness of the population. Normalized fitness has three characteristics being that it ranges between 0 and 1, it is larger for better individuals in the population and the sum of the normalized fitness values is always 1.

In terms of implementing a genetic algorithm in a complex digital system simply counting the number of correct responses would be the easiest method of assigning a fitness value for a particular individual. The raw fitness and standardized fitness would also be easy to implement since they are based on addition and subtraction operations. Though advantageous, adjusted fitness and normalized fitness would require the use of division and floating point numbers which would increase the amount of hardware overhead needed for such a simple operation. Therefore adjusted fitness and normalized fitness will not be considered in this work.

2.4 Steady State Genetic Algorithm

The Canonical Genetic Algorithm is flexible enough for modifications to its flow too. Usually, copies of individuals are created in the population using the Canonical approach. It is assumed that the fitness of the population will increase collectively. However, many researchers have developed techniques for removing copies of individuals in order to increase variability in the population and reduce computations of the same chromosome. Such Genetic Algorithms are called Steady State Genetic Algorithms. Steady state genetic algorithms attempt to solve this problem by removing copies of the same individuals in the population. Duplicate checking becomes advantageous when the population size is small, the chromosomes are short or the evaluation time is large.

The Steady State Genetic Algorithm is outlined below [12]:

1. Generate initial population,
2. Evaluate each individual and assign fitness value,
3. Select two individuals without repetition such that the probability of selection of each individual is proportional to its fitness,
4. Pair the individuals randomly to form parents,
5. With a probability, P_c , perform **crossover** on the parents to generate two offspring otherwise clone the parents ($P_c \cong 0.6 - 0.9$),
6. **Mutate** the offspring with a probability P_m otherwise leave the offspring untouched ($P_m \cong 0.001 - 0.3$),
7. Perform **inversion** (mutation on a series of bits) on the offspring with probability P_i if the application requires it,
8. If offspring are duplicates of any other individual already in the population, then reject,
9. Evaluate offspring,
10. If the offspring are better than the worst individuals in the population, then replace the two worst individual with the offspring,
11. Check **termination criteria**. If criteria not met, go back to Step 3, else go to Step 11,
12. Display results.

The one major disadvantage of the Steady State Genetic Algorithm is that it is susceptible to stagnation. If a majority of offspring is inferior, the algorithm rejects them and continues executing more trials on the existing population for long periods of time with no gain in fitness. The population will remain in a localized portion of the search space. *Only the Canonical approach will be investigated in this work. The additional steps required to implement a Steady State approach would increase the amount of hardware overhead. Once the Canonical approach has been investigated thoroughly, future modifications to this method can be assessed in terms of effectiveness over the basic evolutionary approach.*

Since the stages of the Canonical Genetic Algorithm are synchronous and the chromosome are essentially binary valued bit vectors, extending Genetic Algorithms into the digital domain would be a practical endeavor. Efforts have been made to extend this approach to the design of hardware systems, integrating with Evolvable Hardware research. Some researchers have tried to develop Genetic Algorithms on Field Programmable Gate Arrays (FPGA) where the Genetic Algorithm can be described in a hardware description language and downloaded into the configurable chip. The next step of this research would be to create a stand-alone Genetic Algorithm subsystem such that when activated it could be used to re-configure another part of the FPGA chip.

Chapter Three: Adaptive Hardware System

Evolvable Hardware embraces all optimization problems involved in the hardware design process, but the ultimate goal is the development hardware systems with on-line adaptation capabilities. Applications requiring optimum performance due to changing environments can be integrated with an evolutionary component capable of fine-tuning system parameters during real-time operation. Development of such Adaptive Hardware Systems will require the development of autonomous subsystems capable of performing adaptation processes without external computing resources.

3.1 Adaptive Hardware Approaches

Though the most current research in Evolvable Hardware is focused on developing the intrinsic and extrinsic approaches further, [17][19] ideas aimed at the development systems with On-Line Adaptation capabilities have just begun to surface [22][23]. Torresen attempted to identify Field Programmable Gate Array (FPGA) based hardware architectures that could support adaptive hardware systems. Torresen [22] categorized architectures into three different degrees of hardware adaptivity for configurable computing, namely, Context Switching, Static, and Dynamic. In Context Switching the FPGA contains a set of configurations, which it switches between. These configurations could simply provide computation speed up or represent a set of "adaptations" which the device selects at runtime to best suit the environment. In the Static approach, the configuration in the FPGA is the same throughout the lifetime of the system, meaning there is no adaptivity at runtime. Dynamic systems allow for parts of the configuration to be changed from time to time. It is the most suitable approach for integration with Evolvable Hardware schemes.

Torresen went on to propose various hardware architectures for implementation of these approaches. Context switching would involve the use of User Defined Configuration Registers (UDCR) that could be used to program the configurable hardware. Each configuration is attached to a MUX so that only one configuration could be selected at a time through an FPGA configuration register. Static adaptive hardware approaches could be used in applications involving large amounts of data to be processed as either information or templates for matching. In this architecture two shift registers are used as interchangeable loading and matching buffers. The set of templates are stored in external RAM and loaded into a template register. A controller synchronizes the process of loading and matching the data to a set of external templates. Dynamic adaptive hardware approaches would involve the integration of a static subFPGA in charge of manipulating dynamic subFPGAs. The static subFPGA would be responsible for

loading new configurations into the dynamic subFPGAs when they are not in operation. This dynamic approach could be translated directly into the Evolvable System-On-A-Chip approach to Evolvable Hardware.

The Adaptive Hardware System presented in this work consists of an Evolutionary Adaptation System, Configurable Working System, and On-line Monitoring and Diagnostic System. The On-line Monitoring and Diagnostic System monitors the output data and when it notices that output data is incorrect or changes in the systems environment occur, it signals the Evolutionary Adaptation System to commence operation and places the system in adaptive operation mode. The Evolutionary Adaptation System begins applying each configuration chromosome to the configurable unit. The Configurable Working System must be designed such that normal operation is not disturbed during the adaptive process. When the Autonomous Evolutionary Algorithm finds an optimum solution it signals back to the On-line Monitoring and Diagnostic System that it has completed, flags the Configurable Working System to use the best configuration as the main unit. The On-line Monitoring and Diagnostic System places the system back in to normal operation. When the system is in normal operation one of the configurable system simply processes input data and converts it to output. Figure 3.1 illustrates the system architecture needed to realize an adaptive hardware environment. The Evolutionary-based Adaptation System and On-Line Monitoring and Diagnostic System fit into Torresen's Dynamic model as the static portion of the subFPGA. The Configurable Working System acts as the dynamic subFPGA capable of modification during operation.

Torresen [21] suggested on-line adaptation in real-time would require two parallel configurable units. The primary unit would be applied for normal runtime operation while the second unit would continue to evolve and improve in parallel. The On-Line Monitoring and Diagnostic System would recognize when the secondary unit outperforms the primary unit and exchange them. In this work, the Configurable Working System is considered as only a giant functional block encompassing any kind of internal design. The Configurable Working System could be designed to have many parallel configurable units or just one single unit depending on the nature of the application. If just one single unit were being used in the Configurable Working System, the system would consist of two modes of operation, adaptive and normal. Adaptations could occur during the intervals between input sampling rates. This would give the appearance of on-line adaptation in real-time, though in theory it is not.

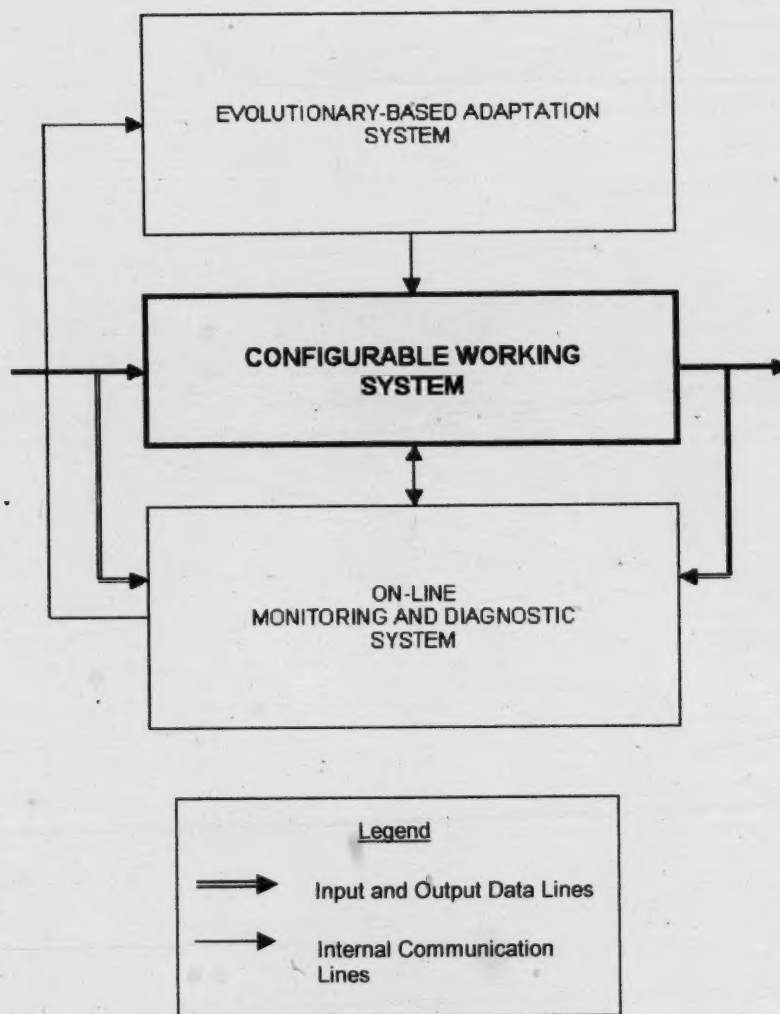


Figure 3.1 On-line Adaptive Hardware System

3.2 Realization of Evolutionary-Based Adaptation System

The first necessary step in realizing this On-line Adaptive Hardware System is the design of an autonomous Genetic Algorithm component. The goal is to develop a hardware-based evolutionary system capable of operating independently of an external fitness function. Modular architecture of the Genetic Algorithm will ensure its ease for modifications and suitability for different applications. Though the interpretation of the binary chromosome will vary from one optimization problem to another, the manipulation of the chromosomes using reproduction operators such as crossover and mutation will stay consistent. The Autonomous Genetic Machine (AGM) presented in this work, manipulates the population and performs initialization, selection, reproduction, population replacement and checks on the termination criteria.

The Autonomous Genetic Machine accepts its operating parameters and begins the evolutionary process when it receives a Start signal. The AGM stores and manipulates the population and associated fitness representation internally in RAM by executing a Canonical Genetic Algorithm. When it is time to evaluate the population, each chromosome is sent to an External Evaluation Component. The External Evaluation Component receives the chromosome and a data valid signal, processes the individual and returns the fitness value along with a data valid signal to the AGM. So long as the interface requirement is met, the External Evaluation Component can be virtually any optimization problem that can be realized in hardware.

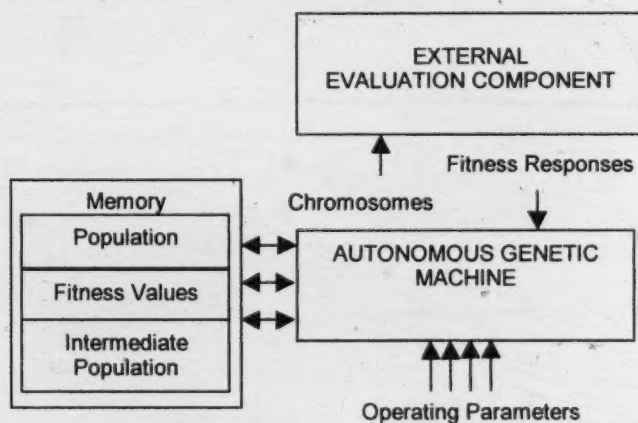


Figure 3.2 Concept of Autonomous Genetic Machine

The External Evaluation Component can be extended for use in optimization problems that require cycling through input and output data. A Fitness Evaluation Subsystem (FES) can be designed to meet the interface requirements of the AGM. This extension to the AGM can make the AHS an even more versatile approach. When activated by a valid chromosome, the FES

cycles through the training data and reads the responses from the application comparing them to the target responses. FES stores this data in two internal ROM components. The application responses for a particular chromosome are characterized as fitness value that is sent back to the AGM. Figure 3.3 illustrates the interface and system architecture for optimization problems using a simple fitness function and how the Fitness Evaluation Subsystem can be used as an extension to the AGM.

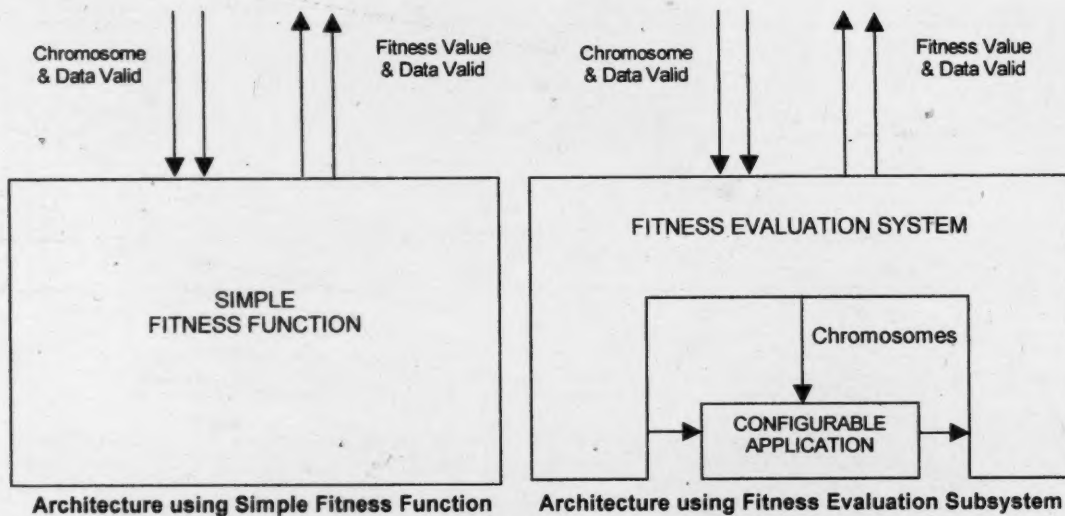


Figure 3.3 System Configurations for External Evaluation Component

This study will attempt to examine the development of the Autonomous Genetic Machine and verify its functionality pertaining to problems involving numerical optimization and linear regression. Three different polynomial functions with diverse topologies will be used to assess the correctness of the AGM in numerical minimization and maximization problems. The AGM will be extended to linear regression problems using the Fitness Evaluation Subsystem. Linear regression problems will involve the creation of a digital circuit representing the generalized function for linear relationships. The Autonomous Genetic Machine will be designed to be platform independent so that it can be synthesized for any FPGA. Performance measurements for various FPGA chips will be presented and analyzed. It should be noted that this system is being developed with one unit as the Configurable Working System in mind. Extending this work to the development of a complete On-Line Adaptive System would require the system to run in adaptive and normal modes such that adaptations occur to the Configurable Working System during the input sampling intervals.

Chapter Four: Autonomous Genetic Machine

Our objective is to conceptualize and develop hardware design architecture for a general-purpose flexible and autonomous Genetic Algorithm engine – an Autonomous Genetic Machine (AGM). A sequential Canonical Genetic Algorithm architecture in VHDL is designed such that an external pipelined architecture can be used when assigning the fitness value for each individual chromosome. Simulation data for Field Programmable Gate Array (FPGA) chips from various manufacturers are examined. Performance measurements of the hardware based Genetic Algorithm relating to chromosome length, population size, fitness representation and system clock frequency are investigated.

This section concentrates on the design architecture for the realization of an adaptive hardware system for synthesis on FPGAs. The Adaptive Hardware System (AHS) will be broken into two main components. The first, being the static portion of the Genetic Algorithm, manipulates the population and performs the main steps of the Canonical Genetic Algorithm. The second component consists of the application that assesses the phenotype of a given potential solution and characterizes its fitness. It will be shown that the static portion of the AHS called the Autonomous Genetic Machine can be used in numerical optimization problems.

4.1 Concept of Autonomous Genetic Machine

The Adaptive Hardware System is designed in this study such that the static Genetic Algorithm portion can be left intact while the application changes provided that the external interface requirements are met. Though the interpretation of the binary chromosome will vary from one optimization problem to another, the manipulation of the chromosomes using reproduction operators such as crossover and mutation will stay consistent; the Genetic Algorithm is broken into two main components. The first being the portion of the Genetic Algorithm that manipulates the population and performs initialization, selection, reproduction, population replacement and checks on the termination criteria. The second component is the External Evaluation Component that applies each individual chromosome to the optimization problems and calculates its fitness value. Figure 4.1 illustrates the two main components of the Adaptive Hardware System, as in Figure 3.2, with specification of operating parameters.

The Autonomous Genetic Machine accepts the threshold values for crossover and mutation, the maximum number of generations and the seed value for the random number generator. It begins the evolutionary process when it receives a Start signal. The AGM stores and manipulates the population and associated fitness representation internally in RAM. When it is time to evaluate the population, each chromosome is sent to the Evaluation Component.

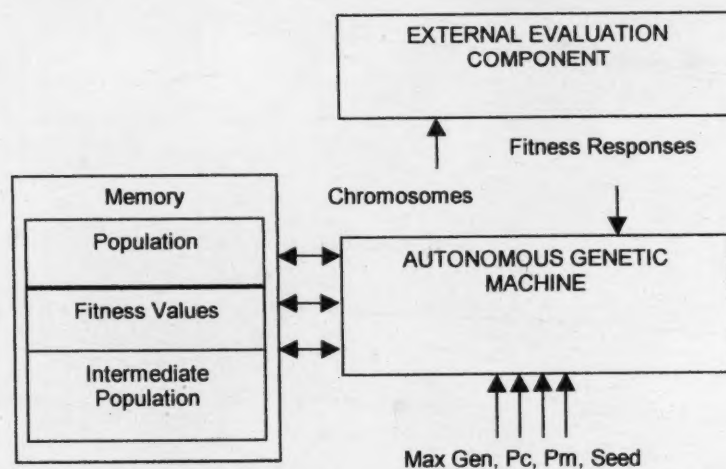


Figure 4.1 Concept of Autonomous Genetic Machine

The External Evaluation Component receives the chromosome and a data valid signal, processes the individual and returns the fitness value along with a data valid signal to the AGM. So long as the interface requirement is met, the External Evaluation Component can be virtually any optimization problem that can be realized in hardware. In the case of simple polynomial optimization problems, the chromosome can be simply processed through a series of pipelined multipliers, adders. Any constraints or penalties can be applied to the chromosome inside the External Evaluation Component.

4.2 Design of Autonomous Genetic Machine

The AGM systems consist of small internal subsystems that communicate with one another and manipulate the population stored internally on the FPGA. An internal controller in each component monitors the system progress and synchronizes the execution of these subsystems. The memory components are implemented internally on the FPGA using embedded memory.

The main purpose behind this design is to create a Genetic Algorithm for hardware synthesis that can be ported to any kind of FPGA. Since VHDL allows the designer to write the hardware implementation behaviorally, all components in the Genetic Algorithm are written as state machines using VHDL. Though there is a sacrifice in performance compared to components that are described structurally, the use of state machines allows for easy modification and understanding versus an implementation that is described structurally.

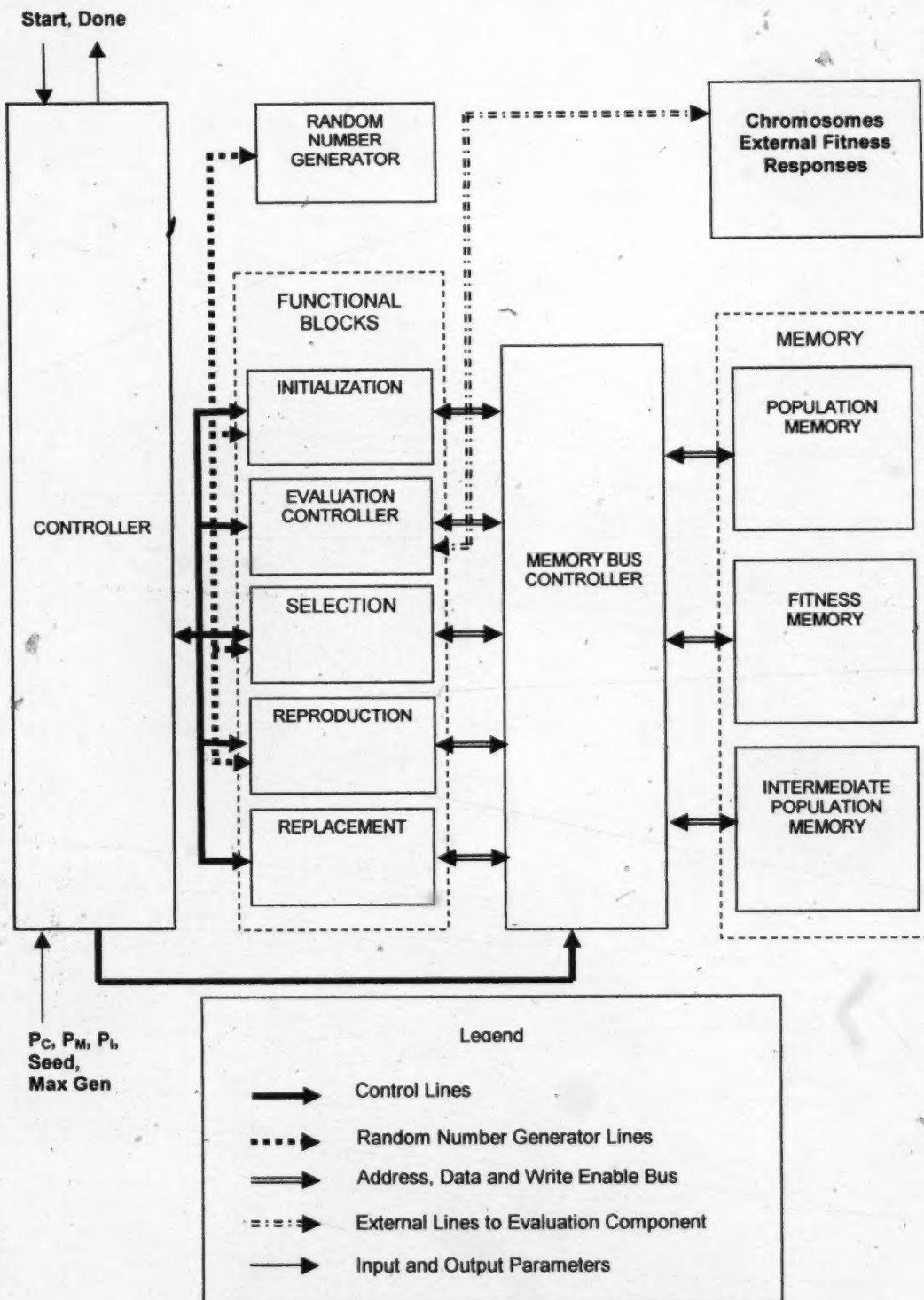


Figure 4.2 Architecture for Autonomous Genetic Machine

4.2.1 High-Level Architecture

The Autonomous Genetic Machine (AGM) is divided into seven major components that integrate into the flow of the Canonical Genetic Algorithm illustrated in Figure 4.2.

1. Initialization
 - Generates random initial population
2. Evaluation Controller
 - Sends each individual to the evaluation components and receives fitness value
3. Selection
 - Selects fittest individuals using a binary tournament method
 - Pairs the individuals randomly to form parents
4. Reproduction
 - With a probability, P_C , performs *crossover* on the parents to generate two offspring otherwise clones of the parents ($P_C \cong 0.6 - 0.9$)
 - *Mutates* the offspring with a probability, P_M , otherwise leaves the offspring untouched ($P_M \cong 0.01 - 0.3$)
 - Perform *inversion* (mutation on a series of bits) on the offspring with probability P_I if the application requires it.
5. Population Replacement
 - Replaces all the individuals of the previous generation with the parents and offspring
6. Controller
 - Checks *termination criteria*: If criteria is not met, continues to synchronize AGM processes; else stops AGM
7. Pseudo Random Number Generator (LFSR)
 - Sends random bit vectors to Initialization, Reproduction and Selection components when required

Each component, except the Controller, communicates with internal memory components on the FPGA, through a multiplex bus. There are three memory components used:

Population Memory – stores the individual chromosomes

Fitness Value Memory - stores each chromosome's fitness value respectively

Intermediate Population Memory – stores parents and children during parent selection and reproduction

Separating the main population and intermediate population ensures that main population is not corrupted in anyway during parent selection and reproduction. As opposed to dividing one RAM component into byte, word or long word boundaries, using two separate RAM components to store the population and fitness values allows the designer to simply associate the memory address of the individual chromosome in the population memory with the individual's location in the memory component storing the fitness values.

4.2.2 Controller

The Controller is responsible for keeping track of activated components and controlling memory access for each activated component. The functional components are divided such that there are no communication interactions between each other. The only signals they internally receive and send, are REQUEST and DONE to the Controller. When it receives a signal to start, it activates the Initialization Component by sending a REQUEST signal. Also, it sets the appropriate select signals to the Memory Bus Controller so that the Initialization Component can access RAM. When the Initialization Component signals DONE back to the Controller, the Controller deactivates the memory access for the Initialization Component, sends a REQUEST signal to the Evaluation Controller, and activates memory access for the Evaluation Controller. This process repeats for the Tournament Selection, Reproduction and Population Replacement components. Then, the Controller checks if the maximum number of generations has been reached. If the termination criterion is not met, it activates the Evaluation Controller and repeats the process again. If the termination criterion is met, the Controller deactivates all components and their memory access, and sends a done signal indicating the optimization process has completed. The Controller does not communicate with the Pseudo-Random Number Generator. Any component that requires random numbers communicates with Pseudo-Random Number Generator itself.

The Controller synchronizes AGM processes as follows:

STATE CLEAR:

- Disable all access to the Memory Bus Controller
- Set Generation Count to 0
- If GA_START signal received
 - NEXT STATE is INIT
 - Send REQUEST signal to the Initialization Component
- Else NEXT STATE is CLEAR

STATE INIT:

- Enable access to the Memory Bus Controller for the Initialization Component
- If INIT_DONE signal received
 - NEXT STATE is EVALUATE
 - Send REQUEST signal to the Evaluation Controller
- Else NEXT STATE is INIT

STATE EVALUATE:

- Disable access to the Memory Bus Controller for the Initialization Component
- Enable access to the Memory Bus Controller for the Evaluation Controller
- If EVALUATE_DONE signal received
 - NEXT STATE is PARENT SELECTION
 - Send REQUEST signal to the Tournament Arbiter
 - Else NEXT STATE is EVALUATE

STATE PARENT SELECTION:

- Disable access to the Memory Bus Controller for the Evaluation Controller
- Enable access to the Memory Bus Controller for the Tournament Arbiter
- If SELECTION_DONE signal received
 - Send REQUEST signal to the Reproduction Component
 - NEXT STATE is REPRODUCE
 - Else NEXT STATE is PARENT SELECTION

STATE REPRODUCE:

- Disable access to the Memory Bus Controller for the Tournament Arbiter
- Enable access to the Memory Bus Controller for the Reproduction Component
- If REPRODUCE_DONE signal received
 - Send REQUEST signal to the Population Replacement Component
 - NEXT STATE is POPULATION REPLACE
 - Else NEXT STATE is REPRODUCE

STATE POPULATION REPLACE:

- Disable access to the Memory Bus Controller for the Reproduction Component
- Enable access to the Memory Bus Controller for the Population Replacement Component
- If REPLACE_DONE signal received
 - NEXT STATE is CHECK TERMINATION
 - Else NEXT STATE is POPULATION REPLACE

STATE CHECK TERMINATION:

- Disable access to the Memory Bus Controller for the Population Replacement Component
- If Generation Count > Maximum # of Generations
 - NEXT STATE is FINAL EVALUATION
- ELSE
 - NEXT STATE is EVALUATE
 - Increment Generation Count

STATE FINAL EVALUATION:

- Enable access to the Memory Bus Controller for the Evaluation Controller
- If EVALUATE_DONE signal received
 - NEXT STATE is FINISH
 - Send REQUEST signal to the Tournament Arbiter
 - Else NEXT STATE is FINAL EVALUATION

STATE DONE:

- Disable access to the Memory Bus Controller for all components
- NEXT STATE is CLEAR

4.2.3 Memory Bus Controller

The Memory Bus Controller consists of a set of three multiplexors for each of the three RAM components. The first multiplexor controls access to the address lines for the associated RAM component. The second multiplexor controls access to the data lines, and the third component controls access to the write enable line. Address, data and write enable signals from all functional components feed into each multiplexor. The select lines on each multiplexor permit memory access for the respective functional components, and are set by the Controller synchronizing the evolutionary process.

4.2.4 Initialization

The Initialization Component creates random chromosomes and stores the data in the Population RAM. When activated, it receives random bit vectors of chromosome length from the Pseudo-Random Number Generator and simply sends the data off to memory until it has reached the maximum address in the Population RAM.

4.2.5 Evaluation

When activated, the Evaluation Controller simply reads all chromosomes in the Population RAM, sends each individual to the external evaluation component and writes the corresponding fitness value to the appropriate memory location in Fitness RAM. After all chromosomes have gone through this process the Evaluation Controller signals that it has completed execution. The Evaluation Controller has two components used to interface with the External Evaluation Component called the Evaluation Sequencer and the Fitness Writer. The Evaluation Sequencer is responsible for retrieving each chromosome from Population RAM and sending them out to the External Evaluation Component. The Fitness Writer receives a fitness value for the individual from the External Evaluation Component and writes the value to Fitness RAM.

The Evaluation Controller can communicate with virtually any kind of External Evaluation Component so long as its interface requirements are met. Figure 3 illustrates how the Evaluation Controller communicates with the External Evaluation Component. The Evaluation Sequencer sends the chromosome along with a "data valid" signal to the External Evaluation Component and the Fitness Writer receives a corresponding fitness value along with another "data valid" signal from the External Evaluation Component.

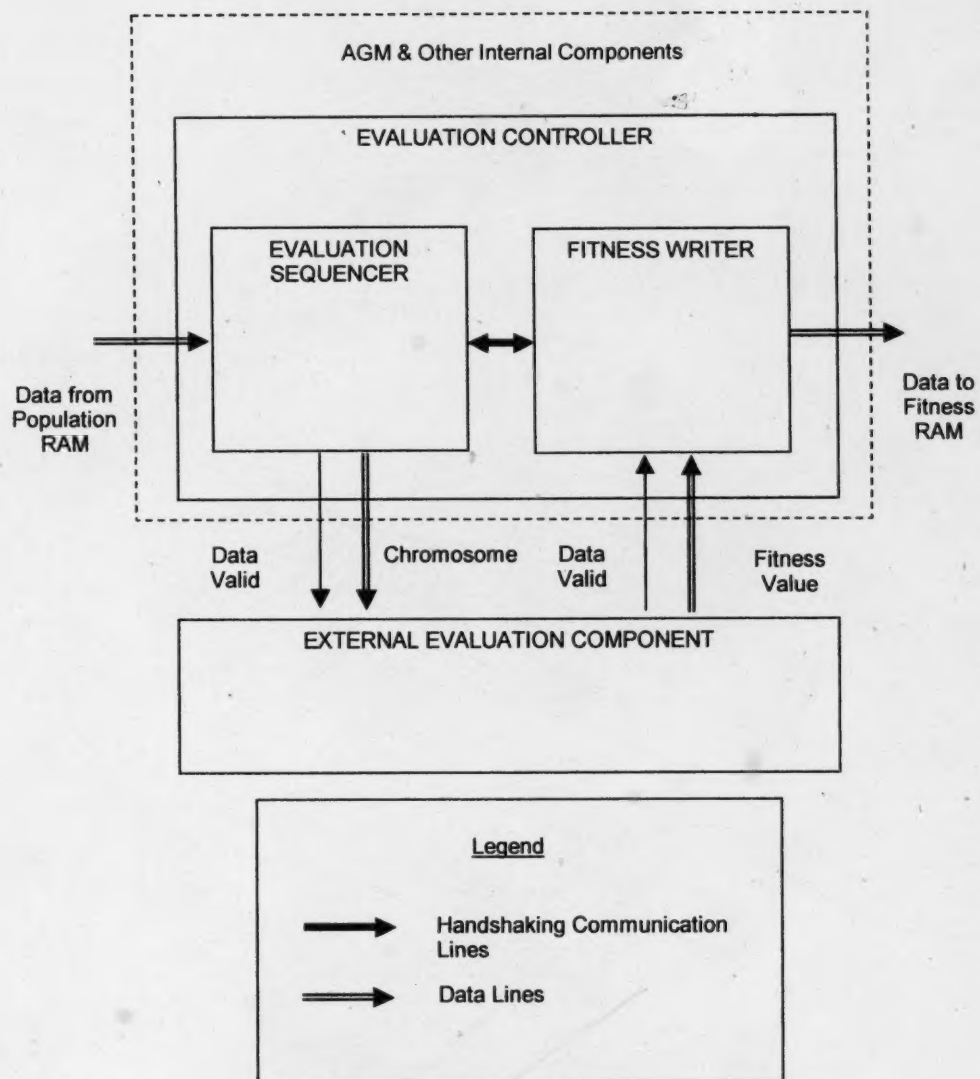


Figure 4.3 Architecture for Evaluation Controller

4.2.6 Selection

The selection method used in this design is Tournament Selection due to its simplicity. Tournament Selection involves random selection of candidate parents and comparison of associated fitness values. Roulette Wheel or Stochastic Universal Selection involves cycling and adding the fitness values of all members of the population while performing comparisons based on random threshold value(s). When the threshold is exceeded, the individual that broke the threshold is selected as a parent for reproduction. Thus, Tournament Selection would require less hardware overhead yielding the quickest execution time.

Tournament Selection is broken into two components, the Candidate Selector and the Tournament Arbiter. The Candidate Selector communicates with the Pseudo-Random Generator to get two random Population RAM addresses and their associated fitness value in Fitness RAM. The information passes on to the Tournament Arbiter, which controls when the Candidate Selector is activated, compares the two fitness values and selects the best parent's address in Population RAM that yielded the better fitness value. Then, it retrieves the chromosome of the winning candidate from Population RAM and writes the individual to the Intermediate Population RAM. This process continues until the lower half of the Intermediate Population RAM is reached.

4.2.7 Reproduction

Reproduction is broken into two components, the Probability Selector and the Reproduction Component. The Probability Selector obtains randomly generated probabilities for crossover, and mutation, two parents from the Intermediate Population RAM and a random crossover point. The Reproduction Component compares the probability of crossover to the threshold value P_C . If the probability is less than P_C , the parents are sent to a Crossover component that performs one-point crossover and produces two offspring. If the probability is greater than P_C , the parents are copied as offspring, analogous to a clone of the parents. One of the two offspring is randomly selected as a valid child and the reproduction process continues. The probability of mutation is compared to the threshold value P_M . If the probability of mutation is less than P_M , one bit of the selected child is mutated. If the probability of mutation is greater than P_M , the offspring is left alone. Inversion, mutation on a series of bits in the chromosome, is performed on the offspring if required. In the last stage, the child chromosome is sent to the Intermediate Population RAM. This process continues until the upper half of Intermediate Population RAM is filled.

4.2.8 Population Replacement

This component simply reads data from Intermediate Population RAM and writes over the chromosomes in Population RAM. In the Genetic Algorithm implementation presented in this paper, the best individuals are not saved and propagated into the next generation. It is assumed that as fitter individuals are found, the overall fitness of the population will increase collectively.

4.2.9 Pseudo-Random Number Generation

Pseudo-random number generation is obtained by using a standard Type II Linear Feedback Shift Register (LFSR) algorithm [16]. The LFSR is designed for bit vectors ranging from 2 to 32. Taps are obtained based on the number of bits used in the pseudo-random sequence and the minimum number of taps that sequences through every possible 2^n values before returning to the initial seed value without repetition.

4.3 Hardware Implementation of Evolutionary Environment

The main purpose of this design is to develop a Genetic Algorithm for hardware synthesis that can be ported to any kind of FPGA. Since VHDL allows the designer to write the hardware implementation behaviorally, all components in the GA were written as state machines. Though there is a sacrifice in performance compared to components that are described structurally, the use of state machines allows for easy modification and understanding versus an implementation that is described structurally. A description of the interface of each component in the Genetic Algorithm is provided.

4.3.1 Memory

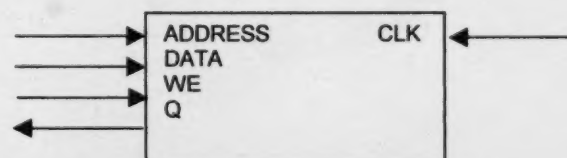


Figure 4.4 Interface for Memory Component

The memory components are synchronous memory with separate input and output ports. The write enable signal indicates a read or write transaction. Write operations take one clock cycle to execution while read operations take two clock cycles to execute. This property creates a lot of wait states in subsequent components that read from RAM, which leaves room for performance optimizations in the design.

4.3.2 Controller

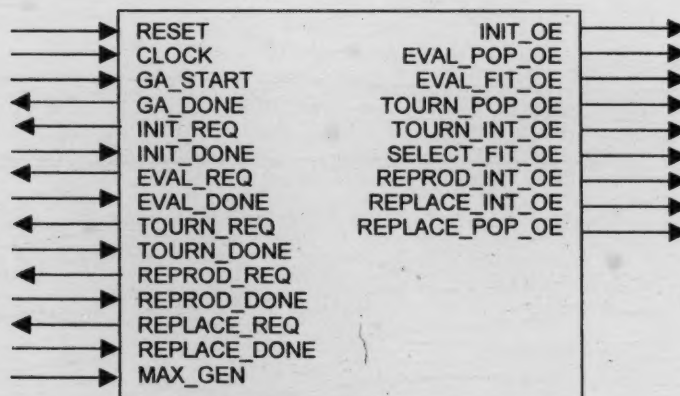


Figure 4.5 Interface for Controller

The Genetic Algorithm controller contains a single GA_START input signal and a single GA_DONE output signal. Also, there are output signals for requesting activation of the various components and input signals for each component that indicate when an operation is complete. Memory control output enable signals for access to the multiplex bus are synchronized with the current component that is activated through the signals suffixed with "OE". The termination criterion, being the maximum number of generations, is an input to the controller. The maximum number of generations possible is set at 1024 because a 10-bit counter is used internally to increment the current generation. A 10-bit counter is used because it is not big enough to significantly slow down the maximum clock speed of the entire system, a maintains the ability to perform this optimization process for a reasonable number of iterations.

4.3.3 Initialization



Figure 4.6 Interface for Initialization Component

The Initialization component is activated with an INIT_REQ signal. Then, the RAND_ENABLE signal is sent to the pseudo-random number generator and random bit vectors (RAND_DATA) are obtained as input to the initialization component. Address, data and write enable lines are used to write the random chromosomes to Population RAM. This process repeats until the Population RAM is filled. Finally, the INIT_DONE signal goes high signaling that the initialization process is complete.

4.3.4 Evaluation Controller

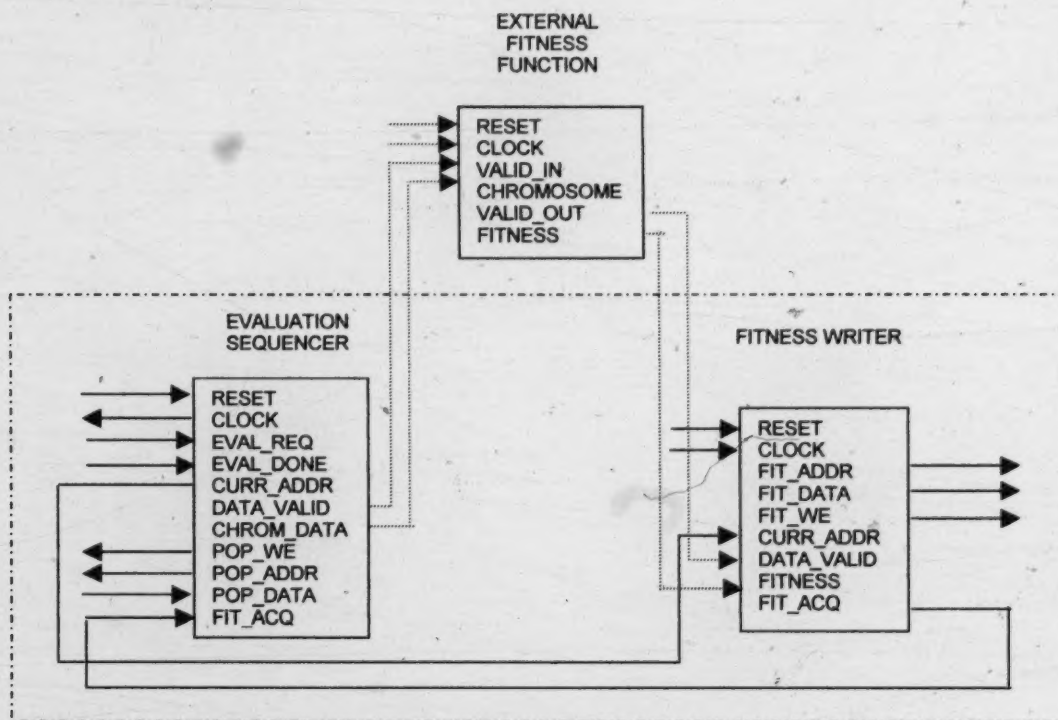


Figure 4.7 Interfaces for Internal Components in Evaluation Controller

The Evaluation Controller consists of two components, the Evaluation Sequencer, and Fitness Writer. When the Evaluation Sequencer receives an EVAL_REQ signal, it cycles through the population RAM (using the POP_ADDR, CHROM_DATA and POP_WE lines) and sends each chromosome through an Evaluation Pipeline. The External Fitness Function receives the current CHROMOSOME, and DATA_VALID signals, in order to calculate the fitness. The corresponding address for each individual chromosome is passed to the Fitness Writer. When the Fitness Writer receives valid data it takes the result from the external fitness function and writes the information to the Fitness RAM based on the current address lines supplied by the last stage of the Evaluation Sequencer. Then it signals FIT_ACQ back to the Evaluation Sequencer to let it know the fitness value has been processed. When the Evaluation Sequencer receives a FIT_ACQ signal for the final chromosome it sends an EVAL_DONE signal back to the Controller.

4.3.5 Selection

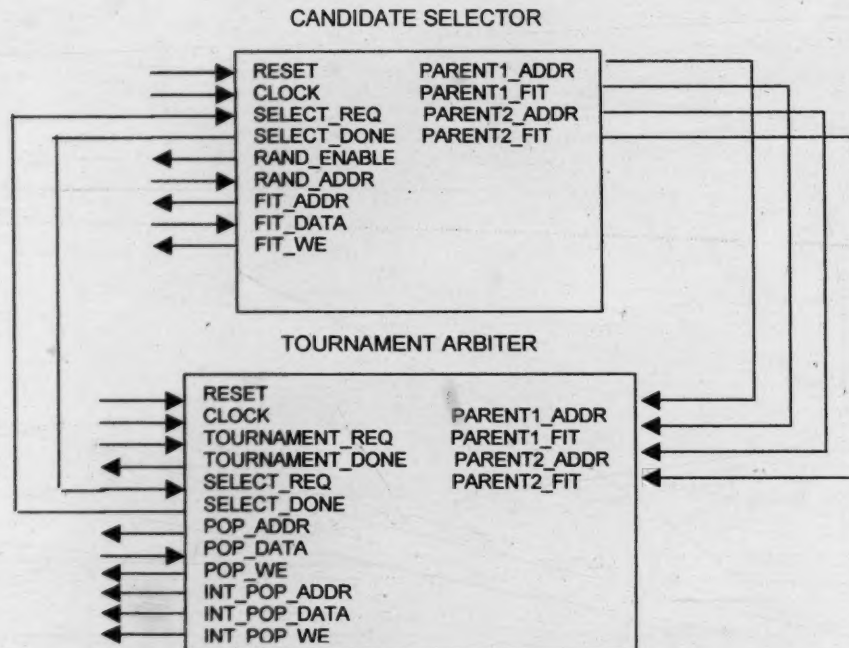


Figure 4.8 Interfaces for Internal Components in Selection Component

The selection component is divided into two main components, the Candidate Selector and the Tournament Arbiter. When the Tournament Arbiter receives a TOURNAMENT_REQ signal from the Controller, it activates the Candidate Selector. The Candidate Selector retrieves two random addresses from the random number generator and the associated fitness values in Fitness RAM. As soon as the fitness values have been extracted from memory, the Candidate Selector signals that the data is ready by setting the SELECT_DONE signal high. The Tournament Arbiter takes the candidates address values and fitness values, compares the two fitness values, retrieves the better chromosome from Population RAM and writes the data to the Intermediate Population RAM. Then the Tournament Arbiter makes another request to the Candidate Selector and the process continues. When half of the Intermediate Population RAM is full of chromosomes, the Tournament Arbiter signals that Tournament Selection is complete by sending a Tournament Done signal back to the GA Controller.

4.3.6 Reproduction

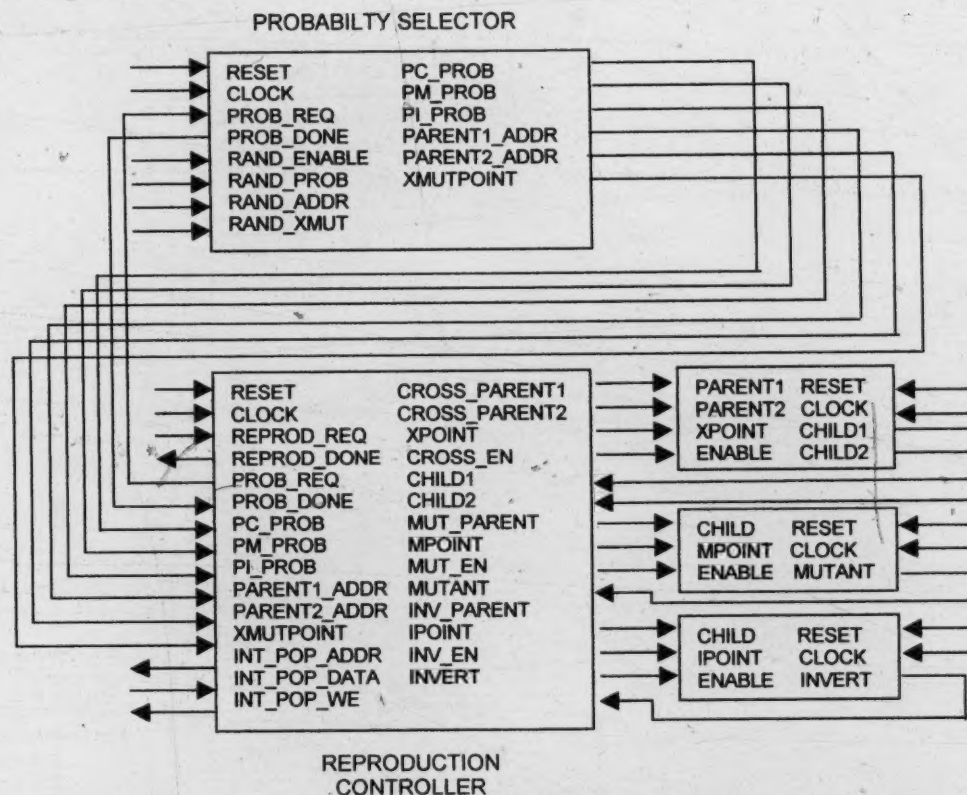


Figure 4.9 Interfaces for Internal Components in Reproduction Component

The Reproduction Component consists of five components, Probability Selector, Reproduction Controller, Crossover, Mutate and Invert. When the Reproduction Controller receives a REPROD_REQ signal, it activates the Parameter Selector via the PROB_REQ signal. The Parameter Selector communicates with the LFSR in order to receive three random probabilities for crossover, mutation and inversion, two random parent addresses from the Intermediate Population RAM and a random crossover/mutation point. When this information has been successfully retrieved, the Parameter Selector signals that the data is ready to the Reproduction Controller via the PROB_DONE signal. The Reproduction Controller retrieves the parent chromosomes from the Intermediate Population RAM. Based on the probability P_c , the Reproduction Controller enables the crossover component and receives two children. The Reproduction Controller randomly selects one of the offspring by performing an XOR operation of the first bit of the random probabilities P_c , P_m and P_i . Based on the probability of mutation, P_m , and inversion, P_i , the Reproduction Controller enables the mutation and inversion components and sends the selected offspring. The final child is written to the Intermediate Population RAM. The Probability Selector is activated again and the process repeats until the other half of Intermediate Population RAM is filled with offspring.

4.3.7 Population Replacement

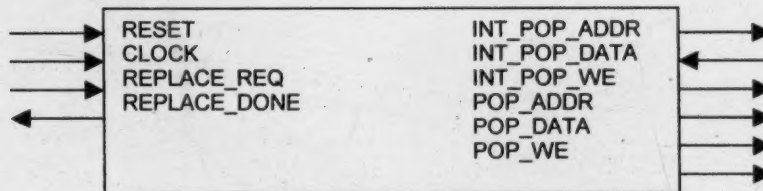


Figure 4.10 Interface for Replacement Component

The Population Replacement component is activated with a REPLACE_REQ signal. Then, it retrieves the chromosomes from the Intermediate Population RAM component via the Intermediate Population address, data and write enable lines and writes the chromosomes to the Population RAM via the Population address, data and write enable lines. When all of the chromosomes in the intermediate population have been written to Population RAM, the REPLACE_DONE signal goes high indicating that the Population Replacement operation is complete.

4.3.8 Pseudo-Random Number Generation

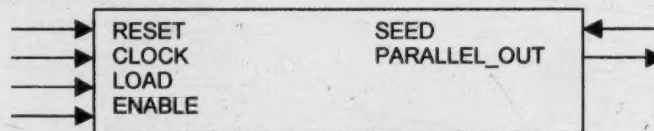


Figure 4.11 Interface for LFSR

The pseudo-random number generator takes in an initial seed value called SEED. When the ENABLE signal goes high, the seed is loaded in and the next random number in the sequence is generated as PARALLEL_OUT. The LOAD signal allows this generic LFSR to be used as a simple register for pipeline usage; the seed is simply copied to the output.

Pseudo-random number generation is obtained by using a generic one-to-many Linear Feedback Shift Register (LFSR) algorithm. The LFSR is designed for bit vectors ranging from 2 to 32. Taps are obtained based on the number of bits used in the pseudo-random sequence and the minimum number of taps that will sequence through every possible value before returning to the initial seed value without repetition. The tapped values are put through an XOR chain combined with the original seed followed by an inverted OR chain. The result is fed back into the register. This method allows for maximal length sequence to consist of all 2^n values including the value

where all bits are logic 0. If a simple XOR chain is used, the maximal length sequence would consist of all $2^n - 1$ values excluding 0 [16].

4.4 Experiments with Numerical Optimization

Simulations of the Autonomous Genetic Machine using three different three-dimensional polynomial functions as the External Evaluation Component are examined in order to verify proper functionality of the system. The objective of the AGM is to find either the minimum or maximum point in each of these functions, illustrated in Figure 12. The three functions used are:

1. $f(x,y) = x^2 + y^2$
2. $f(x,y) = (x^2 + 4x - 16)(y^2 - 5y + 9)$
3. $f(x,y) = ((x^2 - 11x + 30) - (y^2 - 4y - 16)) * (x * y)$

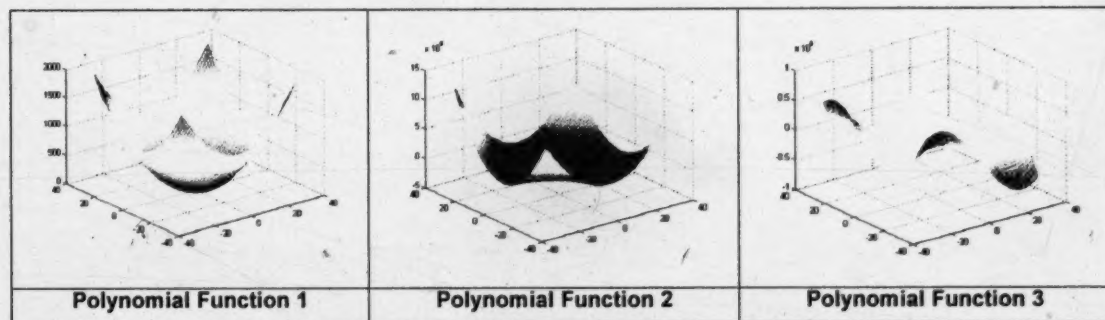


Figure 4.12 Polynomial Functions Integrated with AGM

In the first experiment the objective is to find the minimum of the first polynomial function over the bounds $x=[0,63]$ and $y=[0,63]$, and $x=[-32,31]$ and $y=[-32,31]$, using a population size of 16 individuals and chromosome length of 12 bits representing two unsigned integer values. The probability of crossover, mutation and inversion are varied for a fixed number of generations and the behavior of the AGM is assessed. The seed for the random number generator is obtained by using a program written in C calling it built in random function to obtain random bit values. Table 4.1 summarizes the results obtained from the AGM over the bounds $x=[0,63]$ and $y=[0,63]$ when the system is clocked at 25 MHz.

Final Result	P _c (%)	P _m (%)	Seed	Total Execution Time (msec)	Time to Correct Value (msec)
(0,0)	100	30	0xAD2	2.34024	1.224
(0,0)	100	30	0x97D	2.34136	0.4562
(0,0)	100	30	0xDA1	2.34024	1.503
(0,1)	100	30	0xB7C	2.34024	NA
(0,0)	100	30	0x167	2.33796	0.53080
(0,0)	100	30	0xA75	2.33616	0.57440
(0,0)	100	30	0x3C6	2.33928	1.08466
(0,0)	100	30	0xE5A	2.34100	0.57562
(0,1)	50	5.5	0x62F	2.29376	NA
(4,0)	50	5.5	0x3CE	2.29256	NA
(0,0)	50	5.5	0x4F8	2.28952	0.58670

**Table 4.1 AGM Results for Polynomial Function 1
Over Bounds $x=[0,63]$ and $y=[0,63]$ for 100 Generations**

Since there are 4096 possible points to cover the maximum search space covered by the AGM using a population of 16 individuals for 100 is 1600 points, yielding a maximum possible coverage of 39% in the search space. The maximum possible coverage assumes there are no copies of the same individuals in the population and from generation to generation. If the system is clocked at 25 MHz, 50 generations will have passed in approximately 2.33 milliseconds. Generally, the AGM is finding the correct minimum value within 0.45-1.22 milliseconds, meaning that during the remaining time the population is allowed to saturate with the correct individual. The final result of the AGM is determined by analyzing the final population and calculating its mode, the most common individual. When 100% crossover and 30% mutation is used 7 out of 8 runs produced the correct result. However, the incorrect result (0,1) is in the correct vicinity of the minimum value of Polynomial Function 1. When the probability of crossover and mutation is dropped to 50% and 5%, 1 run out of 3 produced a correct result. However, the other two results of (0,1) and (4,0) are again in the correct vicinity of the minimum value.

The fitness landscape of the first Autonomous Genetic Machine run is examined further to get insights into its behavior. Figure 4.13a,b and c compare the mean fitness of the population with the fitness of the best individual and modal or most common fitness values during various portions of the experiment. During the first 25 generations, the average fitness of the population decreases dramatically, whereas the fitness of the best individual exhibits behavior similar to a decreasing staircase function. From Generation 25 thru 60, the mean fitness of the population fluctuates within 0-160, while the fitness of the best individual continues its decreasing staircase manner. At Generation 60, the best individual is found, yielding a fitness of 0. From Generation 60 to 100, the mean fitness of the population continues to fluctuate while the evolutionary process preserves the best individual.

This difference in mean and minimum fitness values of the population requires further study hence, the modal fitness of the population is examined in order to gain a clearer picture of the population's behavior from generation to generation. During the first 25 generations there is a quite a dramatic decrease in number of similar individuals. This is expected since the AGM starts off with a random population and through the selection and reproduction narrows its search. From Generation 25 to 60, the population saturates with the same individual. However, between Generation 50 and 60, the population saturates quickly with more fit individuals until the correct solution was found. From Generation 60 to 100, the population stays relatively saturated with the correct individual. The fluctuations exhibited in mean fitness of the population occur mainly due to mutations, though the population is saturated with the correct solution. Since the best chromosome is 0x000 and the fitness function is $f(x,y) = x^2 + y^2$, mutating any random bit would cause large fluctuations in mean fitness of the population. This behavior shows that the evolutionary process is still ensured though the optimal solution is found and most of the population is saturated with identical individuals.

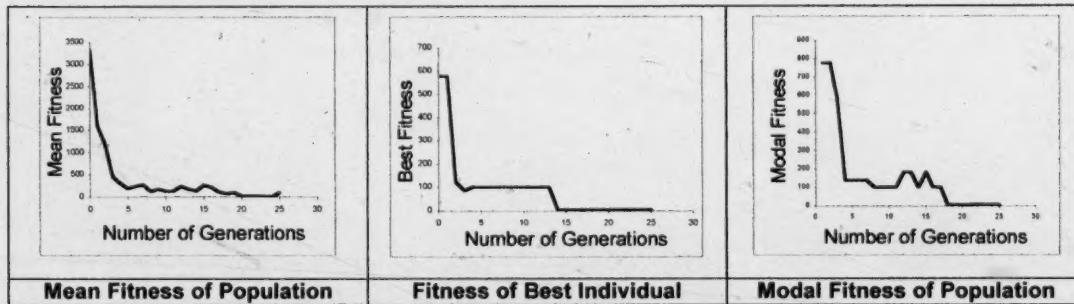


Figure 4.13a. Fitness Landscape for First 25 Generations

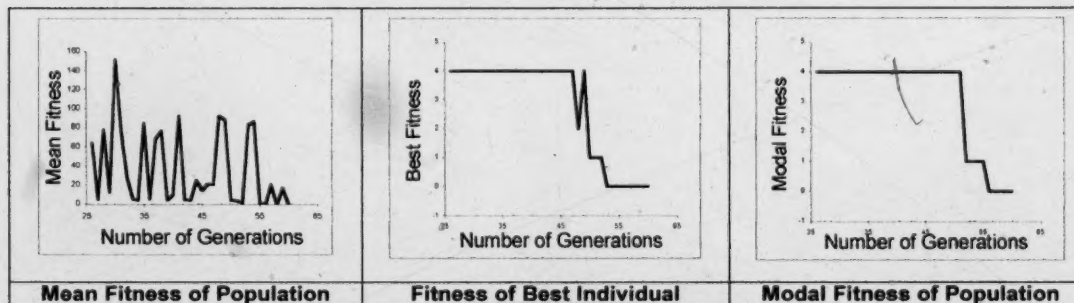


Figure 4.13b. Fitness Landscape for Generations 25 to 60

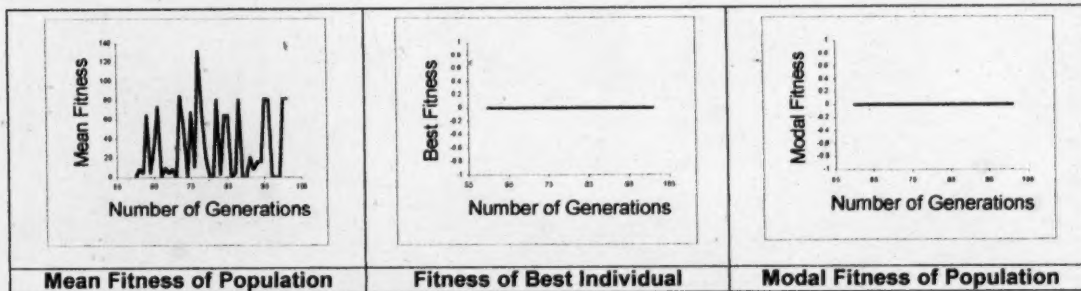


Figure 4.13c. Fitness Landscape for Generations 60 to 100

The second part of the experiment with Polynomial Function 1 is conducted over the bounds $x=[-32,31]$ and $y=[-32,31]$ using a population size of 16 individuals and a chromosome length of 12 bits representing two signed integers. Table 4.2 summarizes the results obtained in this experiment with the system clocked at 20 MHz. Since the first part of this experiment shows that correct results could be obtained in approximately half the time, the number of generations was reduced to 75.

Final Result	P _c (%)	P _m (%)	P _i (%)	Seed	Total Execution Time (msec)	Time to Correct Value (msec)
(-1,-1)	100	30	0	0x1B3	2.38735	NA
(0,-1)	100	30	0	0xF7E	2.38730	NA
(0,-1)	100	30	0	0xCD2	2.38884	NA
(0,0)	100	30	5.5	0x306	2.39003	0.21835
(0,0)	100	30	5.5	0xA95	2.38740	0.28377
(0,0)	100	30	5.5	0xEC5	2.38970	1.78410
(0,0)	60	20	5.5	0x418	2.36135	0.12177
(0,0)	60	20	5.5	0x257	2.35700	0.46125
(0,0)	60	20	5.5	0x49A	2.36130	1.97705
(0,0)	50	12	5.5	0xE1B	2.34855	0.18175
(1,-1)	50	12	5.5	0x65F	2.34755	NA
(0,0)	50	12	5.5	0x1C4	2.34975	0.15150

Table 4.2 AGM Results for Polynomial Function 1 Over Bounds $x=[-32,31]$ and $y=[-32,31]$ for 75 Generations

Using a population size of 16 for 75 generations yields a maximum possible coverage of 29% or 1200 points of the search space. Three runs are conducted using 100% crossover, 30% mutation and no inversion. The final results obtained were in the vicinity of the minimum point. However, no correct answer is obtained.

When inversion is introduced, 8 out of 9 cases produced the correct result. Though each run took approximately 2.35 milliseconds the correct results were obtained between 0.12-1.97

milliseconds. Inversion is kept constant at 5.5% while crossover and mutation was reduced from 100% and 30% to 50% and 12% respectively. When signed integers are used in the chromosome, inversion seems to provide enough variation in the population needed to find the minimum of Polynomial Function 1. This occurs due to the nature of the signed integers in the chromosome. For example, the representation of (-1, -1) is 0xFFFF. If the population is mainly saturated with this individual and the AGM does not use inversion and only mutation then it will be extremely difficult to find the value (0,0). Therefore, by introducing inversion, the AGM is can jump to different portions of the search space, preventing it from getting localized in the negative region only.

Polynomial Function 2 is designed to be more of a challenge for the AGM. There are four different local maximal points at its extremities in the search space. The maximum value of the function is located at (31,-32). Tests are conducted over the bounds $x=[-32,31]$ and $y=[-32,31]$ using a population size of 16 individuals, and a chromosome length of 12 bits representing two signed integers for 50 generations yielding a maximum possible coverage of 20% in the search space. Table 4.3 summarizes the results obtained clocked at 10 MHz. 5 out 18 runs result in the correct value regardless of whether inversion was used or not. However, 6 out of 18 runs yielded results in the correct vicinity of the maximum value. The remaining runs produce results close to the other three extremities of the search space. In this example, inversion does provide some advantage. In the 9 cases where inversion is used 7 cases result either in correct values or results in proximity to the correct maximum value. When crossover and mutation is set to 100% and 30% respectively, 5 out 6 cases result in correct solutions or solutions extremely close to the optimal solution. As the rate of crossover and mutation drop, the AGM obtains solutions in the other extremities of the search space. Overall, the AGM did perform quite well due to the extreme topology of Polynomial Function 2.

Final Result	P _c (%)	P _m (%)	P _i (%)	Seed	Total Execution Time (msec)	Time to Correct Value (msec)
(-32,-31)	100	30	0	0x96A	3.5439	NA
(31,-31)	100	30	0	0xB4C	3.5484	NA
(31,-32)	100	30	0	0x63F	3.5446	0.9656
(31,-30)	100	30	5.5	0x871	3.5477	NA
(31,-32)	100	30	5.5	0xE0D	3.543	2.063
(31,-32)	100	30	5.5	0x452	3.5503	1.3092
(29,29)	60	20	0	0xAC5	3.5607	NA
(31,-32)	60	20	0	0x31E	3.5018	1.7021
(31,31)	60	20	0	0x2D3	3.5014	NA
(31,-24)	60	20	5.5	0x064	3.5074	NA
(26,-24)	60	20	5.5	0x1B7	3.5089	NA
(31,-31)	60	20	5.5	0x78C	3.5116	2.2512
(-23,31)	50	12	0	0xF06	3.4904	NA
(-30,-31)	50	12	0	0x590	3.4869	NA
(-31,31)	50	12	0	0xE1A	3.4879	NA
(29,-30)	50	12	5.5	0xCC5	3.4925	NA
(-30,-32)	50	12	5.5	0xAD2	3.4918	NA
(30,31)	50	12	5.5	0x890	3.4935	NA

**Table 4.3 AGM Results for Polynomial Function 2
Over Bounds $x=[-32,31]$ and $y=[-32,31]$ for 50 Generations**

Polynomial Function 3 is designed to have varying local minima and maxima in order to make things more confusing for the AGM. The function contains two maxima at (-32, -20) and (23, -32) and one global minima located at (-32,23) over the bounds $x = [-32,31]$ and $y = [-32,31]$. Tests for locating the maximum and minimum of the function are conducted using a population size of 16 individuals and chromosome length of 12-bits representing two signed integers for 50 generations clocked at 10 MHz. Probabilities for crossover and mutation are varied from 100% and 30% to 50% and 12% respectively. Inversion is not used. Table 4.4 summarizes the results obtained when the AGM attempts to determine the maximum of Polynomial Function 3. Since there are two optima, half of the results are in the vicinity of the first optima (-32, -20) and the other half in the vicinity of the second optima (23,-32). When the probability of crossover and mutation reached 50% and 12%, the AGM does not perform well. Only 1 out of 3 runs yield results in the correct vicinity of one of the two optimum values. Table 5 summarizes the results obtained when the AGM attempts to determine the minimum of Polynomial Function 3. Again, regardless of the probability of crossover and mutation, 8 out of 9 results are in the correct vicinity of the optimal solution, namely (-32,23). One correct solution is obtained in these 8 runs. The other erroneous result is in the vicinity of local optima located at (31, -15). Thus, the AGM does perform reasonably given that Polynomial Function 3 is designed to be challenging with varying local optima.

Final Result	P _c (%)	P _m (%)	Seed	Total Execution Time (msec)
(25,-32)	100	30	0xCBC	3.7147
(-32,-20)	100	30	0x6D3	3.7101
(-32,-21)	100	30	0xE38	3.7079
(-30,-20)	60	20	0xACE	3.6738
(-30,-12)	60	20	0x86D	3.6659
(22,-32)	60	20	0x791	3.671
(-12,29)	50	12	0x267	3.6548
(31,7)	50	12	0xB46	3.6561
(-31,-15)	50	12	0x1E5	3.661

Table 4.4 AGM Results for Maximum of Polynomial Function 3 Over Bounds $x=[-32,31]$ and $y=[-32,31]$ for 50 Generations

Final Result	P _c (%)	P _m (%)	Seed	Total Execution Time (msec)
(-30,26)	100	30	0x67E	3.7094
(-30,19)	100	30	0xB45	3.7175
(-32,24)	100	30	0x392	3.709
(31,-18)	60	20	0x8F2	3.668
(-28,22)	60	20	0xECA	3.6736
(-32,23)	60	20	0x53B	3.6755
(-31,25)	50	12	0x04D	3.6522
(-32,22)	50	12	0x7C3	3.6559
(-32,15)	50	12	0x9E1	3.6526

Table 4.5 AGM Results for Minimum of Polynomial Function 3 Over Bounds $x=[-32,31]$ and $y=[-32,31]$ for 50 Generations

4.5 Analysis of Autonomous Genetic Machine

Design considerations in the implementation of the AGM for the Flex 10 KE FPGA from Altera and Virtex FPGA from Xilinx are now assessed. The strength of Genetic Algorithms lies in their ability to do a comprehensive random search using only a small population size relative to the size of the search space. The question arises as to what size of population to use when integrating a Genetic Algorithm with a particular problem. The chromosome, associated fitness value and population are the main items on which each component performs operations on; altering the size of these parameters should affect system performance, the number of logic cells, and amount of internally memory cells significantly. Increasing the length of the chromosome increases could increase the delay time for the crossover and mutation operations since more

1-bit comparators and a larger internal counter must be synthesized. Increasing the length of the associated fitness value bit vector could have an impact on the parent selection subsystem because a larger comparator must be synthesized. Increasing the population size causes the size of the address bus to increase. As a consequence, each subsystem needs a larger counter to cycle through memory. The execution time of all functional blocks will increase proportionally with the population size.

Simulations are conducted for the Flex 10 KE EPF10K200SFC672-1 FPGA from Altera and the Virtex xcv1000fg680-4 FPGA from Xilinx. The basic programmable unit in the Altera FPGAs is called a logic cell while specialized embedded logic cells are used to implement memory on the Altera chip. Similarly, Xilinx has a basic programmable unit in their FPGAs called slices. The slices can be configured for a variety of different functions including RAM and contains two configurable logic blocks. The total number of slices available in the Virtex 1000 chip is 12,288.

4.5.1 Population Size and Chromosome Length

Before the AGM is synthesized for a particular FPGA, an important question arises as to what population size it can handle. This number is based on the chromosome length, the fitness representation and the memory resources available on the chip.

Altera chips use Embedded Array Blocks to implement memory. Embedded Array Blocks are groups of embedded cells that can be used to implement RAM, ROM and other combinational circuits. Since RAM is implemented in the FPGA itself, communication with an external RAM chip is not needed. The entire design is self-contained inside the FPGA allowing for ease of experimentation with chromosome length, fitness representation and population size. A single EAB consists of an embedded cell array, with data, address, and control signal inputs and data outputs and can implement a memory block of 256 x 16, 512 x 8, 1,024 x 4, 2,048 x 2, or 4,096 x 1 bits on the Flex 10KE. As more memory is needed multiple EABs are combined to create larger memory blocks. The total number of these logic cells available on the Flex 10 KE chip is 9,984 and the total number of Embedded Array Blocks in the Flex 10 KE chip is 24. Embedded Array Blocks used to implement RAM does not affect the number of logic cells used in the design.

In the Flex 10 KE chip, the total system Embedded Array Block usage can be predicted before simulations. The number of EABs can be predicted knowing that three separated memory blocks are needed for the AGM, being Population Memory, Fitness Value Memory and Intermediate Population Memory. More specifically, we have:

$$\text{Memory\_requirement} = (\text{Fitness Length} + 2 * \text{Chrom. Length}) * \text{Pop. Size}$$

$$\# \text{ EABs} = \text{Memory\_requirement} / \text{Size of EAB}$$

This constraint plus the resulting maximum clock frequency and number of logic cells obtained from the synthesis tool, have to be considered when determining a maximum population size for a given chromosome length and representation of associated fitness value. Figure 4.14a graphically illustrates theoretical and experimental maximum population size permitted for changing chromosome lengths, provided the associated fitness values are represented in 16-bits. Note that there are some differences between these values caused by the synthesis tool, which allows RAM components to increase in powers of two. When the chromosome length is 8-bits, and 16-bits are used to represent the fitness of an individual, then the maximum allowable population size is 3072, however the synthesis tool will allow a population size of 2048 because there are 24 EABs on the Flex 10KE chip and if a population size of 4096 is used, the maximum number of EAB's on the Flex 10KE chip is exceeded.

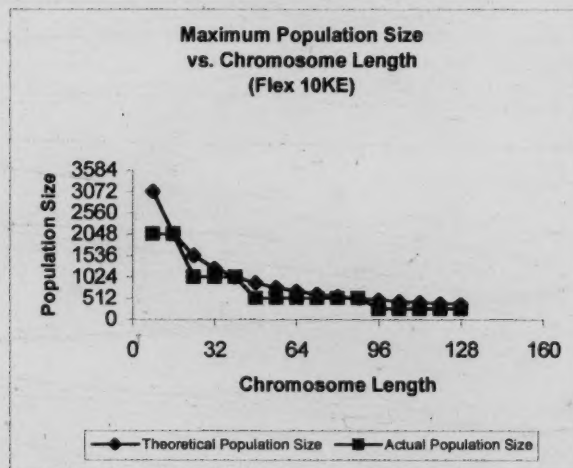


Figure 4.14a. Effects of Chromosome Length on Population for Flex 10 KE FPGA
Fitness Length: 16-bits

The Virtex FPGAs use built-in RAM called Block SelectRAM. Memory can be implemented by describing the memory behaviorally using VHDL and the Block SelectRAM segments will

automatically be arranged to support the model. This way, it is easier to associate one address in RAM with one chromosome rather than using external RAM where memory needs to be divided into bytes and word values. Also, population sizes do not necessarily have to be in powers of two though it is recommended. Figure 4.14b depicts the relationship of the maximum and recommended population size for a given chromosome length. Note that measurements were only taken up to a chromosome length of 64. With a chromosome size greater than 64, the population size is affected not only by the number of available resources in the chip but also by the number of input and output pins for the Virtex chip. The Altera chip had enough external pins to test up to a chromosome length of 128-bits.

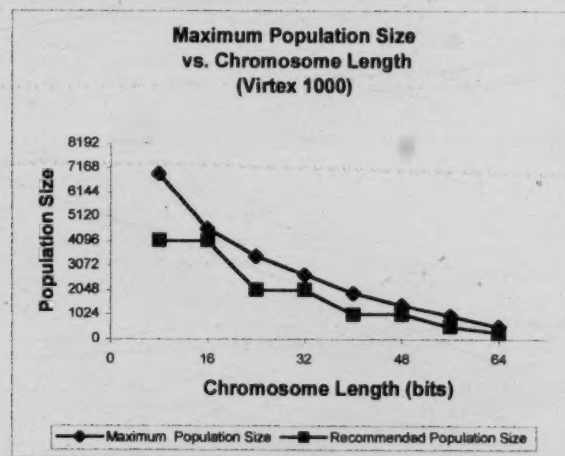


Figure 4.14b. Effects of Chromosome Length on Population for Virtex 1000 FPGA
Fitness Length: 16-bits

Though the basic building blocks in the Altera and Xilinx chips are different, both chips are exhibiting similar behavior with respect to selecting the population size and chromosome length for a given chip. Note that for other FPGAs belonging to the Altera and Xilinx the maximum population size vs. chromosome length graphs will appear similar in topology though the actual numerical values for maximum permitted population size will vary.

In the next set of experiments, the chromosome length and fitness representation is fixed at 8-bits and 16-bits representation while the population size is varied to see the affect on system clock frequency. As the population size doubles, the address width increases by only one bit, therefore, it would be expected that population size would have a negligible affect on the resulting system clock frequency. As the population increased on the Flex 10 KE chip, the system clock frequency fluctuated between 30-40 MHz. However, in the Virtex 1000 chip, the system clock frequency fluctuated between 20-40 MHz. This discrepancy in performance results from the differences in the internal design in both chips. The Altera chips use long, global interconnects between logic

elements call FastTrack Interconnects, whereas, the Xilinx chips use long and short interconnects between logic elements. This will result in varied performance during the synthesis and logic optimization process for each FPGA. In this experiment, the FastTrack interconnects in the Flex 10 KE chip are affected the least by increasing the size of the address bus.

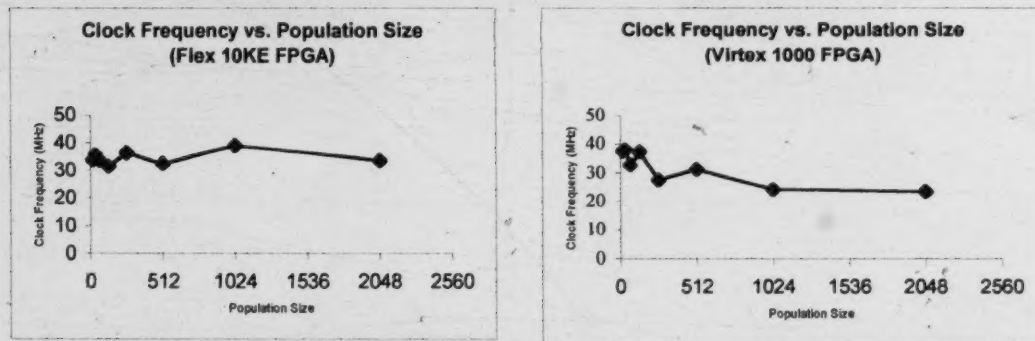


Figure 4.15 Effects of Population Size on System Performance for Flex 10KE and Virtex 1000 FPGAs

4.5.2 Synthesis of AGM

The following section deals with performance issues of the AGM on various FPGAs from different manufacturers. The Table 4.6 and 4.7 summarize the number of basic programmable units used by each component as well as the maximum clock frequency for the Altera and Xilinx chips.

Component	Logic Cells	Clock Frequency (MHz)
Controller	100	43.47
Initialization	47	116.27
Evaluation	72	107.52
Selection	386	35.97
Reproduction	677	35.33
Replacement	63	64.51
LFSR	39	172.41
Total	1343	37.31

Table 4.6 AGM Component Characteristics for FLEX 10 KE
Chromosome Length: 8-bits, Population: 256, Fitness Length: 16-bits

Component	Slices	Clock Frequency (MHz)
Controller	44	90.58
Initialization	33	64.64
Evaluation	29	118.99
Selection	146	46.44
Reproduction	224	40.58
Replacement	24	101.70
LFSR	7	166.22
Total	514	48.31

Table 4.7 AGM Component Characteristics for Virtex 1000
Chromosome Length: 8-bits, Population: 256, Fitness Length: 16-bits

The Virtex chip is clearly a faster and more recent chip, however, in both chips the Selection and Reproduction components have the slowest clock frequency. If speed is an issue these components can be broken down into a pipeline to separate and speed up the operations. Another notable result is in the number of basic programming units used by both the chips for all components. The total number of logic cells required to synthesize the AGM on the Altera chip is 1343 whereas in the Xilinx chip the total number of slices required is 514, keeping in mind that one slice contains two configurable logic blocks.

Even though the clock frequency of the Selection and Reproduction components are 35.97 MHz and 35.33 MHz respectively, the system clock frequency on the Altera chip is 37.31 MHz. On the Xilinx chip the clock frequency of the same components are 46.44 MHz and 40.58 MHz; however, the system clock frequency is 48.31MHz. This occurs due to optimizations performed by the logic synthesizer and place and route tool. If this system is designed as an ASIC, then the total clock frequency could never be greater than the clock frequency of its parts.

In order to assess which chips would be the most ideal for running the AGM on another experiment is conducted where the AGM was synthesized on various non-military grade FPGA chips from different companies. Since the basic unit on each FPGA chip is different the number of look-up tables, flip-flop registers and latches required to synthesize the AGM were used as a basis for comparison. Table 4.8 provides a summary of the vendor, chip, speed grade, number of look-up tables and clock frequency used in this experiment. The number of flip-flop registers and latches needed is 850 and 2 respectively for all chips. On average, the maximum clock

frequency for the AGM synthesized on all chips using a chromosome length of 8-bits, fitness length of 16-bits and a population size 256 is 60 MHz.

Vendor	Chip	Speed Grade	# LUTs	Clock Frequency (MHz)
Altera	FLEX10KE:EPF10K50STC144-1	-3	1342	33
Altera	EXCALIBUR_ARM:EPXA10F1020	C3	1176	62
Altera	APEXII:EP2A70F1508	C9	1253	69
Altera	APEX20KE:EP20K1000EBC652	-3	1253	65
Altera	MERCURY:EP1M350F780	I6	1381	67
Altera	STRATIX:EP1S80F1508	C7	1382	58
Xilinx	VIRTEX:V1000FG680	-6	1082	43
Xilinx	VIRTEX2:2V8000FF1517	-5	1082	69
Xilinx	VIRTEXE:V300EPQ240	-8	1087	85
Xilinx	SPARTAN:S40BG256	-4	1099	60
Xilinx	SPARTAN2:2S200PQ208	-6	1088	131
Xilinx	SPARTAN2E:2S300EFG456	-7	1086	99
Xilinx	SPARTANXL:S40XLCS280	-5	1099	61
Xilinx	COOLRUNNER:R3384XLFG324	-12	3477	25
Xilinx	COOLRUNNER2:2C256F7256	-10	3477	25
Actel	A3200DX:A3200DXVRQ240	-1	3740	32
Actel	ACT2-1200XL:RT1280APG176	-1	3707	22
Actel	ACT3:RT14100ACQ256	-1	3871	32
Lucent	ORCA4E:OR4E8X850HBM680	-2	1015	59
Lucent	ORCA3L:OR3L225BSB432B	-8	1015	61

Table 4.8 AGM Component Characteristics for Various FPGAs
Chromosome Length: 8-bits, Population: 256, Fitness Length: 16-bits

Evolvable hardware applications will benefit greatly since Field Programmable Gate Arrays (FPGAs) are user-programmable and offer significant cost savings over custom manufactured alternatives [3]. Xilinx, the first FPGA manufacturer, uses Static RAM Programming Technology in their FPGAs. These FPGAs use SRAM cells to control user-programmable switches. This feature provides several advantages; the FPGA can be reprogrammed any number of times, programmed in-circuit, fully testable at the factory, and advance quickly to higher capacities. The disadvantage of using SRAM are that SRAM cells consume significant area on the chip and are volatile involving an extra ROM chip to hold the programming bits. Xilinx FPGAs use a two-dimensional array of Configurable Logic Blocks (CLBs), direct wires connecting a CLB to three of its neighbors, general purpose horizontal and vertical routing channels between each CLB, routing switch matrices for the general purpose interconnects, and long lines for high fanout networks requiring low skew like clocks. Each CLB contains a series of look-up tables, multiplexors and flip-flop registers that can provide a wide variety of programmable functionality including read/write RAM. Two CLBs make up the basic unit for Xilinx FPGAs called the Splice. The Xilinx family of FPGAs has the best clock frequency performance ranging from 43 to 131

MHz. In the Virtex family the VirtexE chip gives the best clock frequency, 85 MHz, because the VirtexE chip is based on a six-layer metal 0.22 micron transistors with 0.18-micron interconnects, whereas other Virtex chips are based on a five-layer metal 0.25-micron process giving the VirtexE device better speed grade performance and higher packing density. More recent devices in the VirtexE family have moved to 0.18-micron transistors with 0.15-micron interconnects. The Spartan II chip can be clocked at 131 MHz and outperforms the VirtexE, the Spartan IIE and Spartan XL chips. This occurs because the Spartan II is based on a cost-optimized VirtexE architecture combining six layer metal 0.22-micron transistors with 0.18-micron interconnects, while the Spartan family is based on an older architecture using five-layer metal 0.5-micron transistors with a 0.35-micron interconnects. The Spartan XL devices have 0.35-micron transistors with 0.25-micron interconnects. The Spartan IIE chip provides more advanced built in system and arithmetic features for designing complex hardware systems but does not use faster Configurable Logic Blocks than the Spartan II, therefore performance maybe slightly affected compared to the Spartan II. The CoolRunner chips are based on layers of Complex Programmable Logic Devices (CPLD) that essentially contain an arrangement of multiple Programmable Array Logic (PAL) blocks communicating through a multiplex interconnect bus. The layers of CPLD communicate with each other using an Advanced Interconnect Matrix (AIM). The CoolRunner CPLDs are low-density chips for applications requiring low power and low cost, and thus, have the lowest clock frequency of 25 MHz. The number of look-up tables required to synthesis the AGM ranges from 1082 to 1099 with the exception of the CoolRunner CPLD that uses 3477 gates instead of look-up tables due to its different design architecture [3][28][29][31][32][33].

Altera has developed FPGAs in a slightly different manor. The basic logic cell, called a Logic Element basically consists of one look-up table, special carry circuitry for arithmetic circuits, cascade circuitry for wide logic functions and one flip-flop register that can implement a single function. Each Logic Element is grouped into sets called Logic Array Blocks containing a local interconnect such that any Logic Element can connect to any other Logic Element with the same Logic Array Block. There is also a global interconnect called FastTrack that connects one Logic Array Block to another. The advantage of this chip topology is that the FastTrack can be used for general purpose signals as opposed the long lines in Xilinx chips, that are reserved for signals with high fan out and low skew. All FastTrack wires are identical making interconnection delays predictable, unlike other FPGAs where connections may involve varying numbers of small and long wire segments. The generous connectivity provided by the FastTrack interconnect makes it very easy for CAD tools to configure automatically. The disadvantage of this architecture occurs when circuits require the cascade or carry chain. Since one Logic Element in a Logic Array Block is explicitly linked to one another it may be difficult to successfully map circuits into the device. In the Altera family of FPGAs, the amount of look up tables needed for synthesis marginally varies

from 1176 to 1382. The all of the more advanced chips range in clock frequency from 58 to 69 MHz with the exception of the Flex 10 KE chip at 33 MHz. This occurs due the device characteristics; the Flex 10 KE is built using 0.22-micron process whereas the other chips are built using a 0.13 to 0.15-micron process using all-layer copper interconnects thereby reducing gate delays and boosting performance [26].

The Actel lines of FPGAs are based on a modified CMOS Process Technology called the Antifuse, which are very small programmable elements that are normally open-circuits or high impedance and take on low resistance when programmed. The advantage of using antifuses is that they are non-volatile. However, they are one time programmable. Actel's FPGAs have a general structure that is row-based, with Logic Modules cells arranged in rows. A horizontal routing channel, programmed using antifuse technology separates adjacent rows. Vertical interconnection wires run over the logic cell rows. Actel's Logic Modules consist of multiplexors allowing it to be very small compared to other FPGAs. Actel's ACT 2 and ACT 3 chips comprises of two types of Logic Modules called Combinational and Sequential Modules. The Sequential modules essentially contain the same logic as the Combinational Module with a flip-flop register included. Actel's architecture is different from look-up table based FPGAs because it consists of a large number of small logic cells as opposed to a small number of complex logic cells in Xilinx and Altera FPGAs. In the Actel line of FPGAs the most recent 3200DX, 1200XL and ACT 3 chips are investigated. The 3200DX and 1200XL chips are in essence ACT 2 architectures that contain dual-port synchronous RAM interfaces, FIFO buffers, and JTAG support for digital system testing for applications such as telecommunications, networking and DSP. The ACT 3 chip uses 0.8 micron CMOS technology, enhanced ACT 2 Logic Modules, I/O blocks with flip-flops to reduce "clock-to-output" delays, and more interconnects offering increased speed performance. The ACT3 chip contains circuitry that allows users to implement high-speed complex functions for applications with high computational requirements. The number of Logic Modules needed to synthesize the AGM on ACTEL chips ranged from 3707 to 3871 keeping in mind the design architecture uses large amounts of small logic cells. The Actel chips had performances ranging from 22 to 32 MHz [3][30].

The Lucent ORCA family of FPGAs is similar in design to the look-up table based FPGA architecture used in Xilinx products. The difference lies in the architecture of its basic logic cell, called Programmable Function Units (PFU). Each PFU contains look-up tables that are decomposable, meaning the PFU can be configured as one large 6-input look-up table, or two 5-input look-up tables or four 4-input look-up tables. Thus, small logic functions can be implemented without using a wide range of logic function without consuming an entire PFU. Similar to Xilinx and Altera FPGAs, the ORCA logic cell contains four flip-flop registers and dedicated carry circuitry. Also, the look-up tables can be configured as read/write RAM as in

Xilinx FPGAs. The interconnect architecture between PFUs differs from model-to-model, thereby affecting the speed performance. The number of look-up tables required to synthesize the AGM on the ORCA chips, 1015, was less than both Xilinx and Altera FPGAs. The ORCA chips yielded a speed performance in the same range as the latest Altera chips, being 59 to 79 MHz [3].

A simulation is conducted for the AGM for one generation to provide a mechanism for predicting the execution time for multiple generations. In order to focus more on the performance of the AGM, the evaluation component used to calculate the fitness of each individual is a simple polynomial function designed such that evaluation occurs in a pipelined fashion. Other evaluation components should use a pipelined approach in fitness calculations in order to minimize execution times for one generation when integrated with the AGM. Simulations are conducted with a clock frequency of 20 MHz in order to gauge how long each component takes to complete operation.

Component	Execution Time (μ s)
Controller	-----
Initialization	12.85
Evaluation	102.5
Selection	121.7
Reproduction	180.45
Replacement	51.3
LFSR	-----
Total	468.80

Table 4.9 Performance of AGM
Chromosome Length: 8 bits, Population: 256, Fitness Length: 16, Clock Frequency: 20 MHz

In this simple problem, one generation clocked at 20 MHz takes 468.80 μ sec to execute on the Flex 10 KE FPGA. Therefore, in one millisecond approximately 2 generations will pass. In one second, 2,000 generations will pass. This kind of performance on the FPGA makes it possible for a large number of generations to be executed in a short amount of time if the External Evaluation Component is pipelined effectively. Since Genetic Algorithms generally known to be a time consuming and computationally intensive task, using an FPGA in a communication system with a PC or some other external entity in a specialized hardware system would be a practical endeavor.

4.5.3 Effects of Fitness Representation

The representation of the fitness values for each individual can affect the maximum clock frequency of the entire system. In the Tournament Arbiter, the fitness values of the two candidates are compared. This results in the use of a comparator when synthesized in hardware. If the length of each fitness value is large, the comparator performs the comparison operation on more bits. Thus, the maximum clock frequency of the Tournament Arbiter and the entire system decreases as the size of the comparator increases. The effects of changing the length of the bit vector representing the fitness values have been monitored. The chromosome length is fixed at 8 bits and population size is 256 individuals. Figure 4.16 summarizes the results obtained for the Altera chip. There is an evident relationship between the length of the fitness value and the maximum clock frequency of the AGM. As the fitness value's length increases, the maximum clock frequency of the system decreases almost inversely. Consideration of the representation of the fitness values associated with each chromosome must also be taken into account when applying the AGM to a particular problem on Altera chips. Similarly, the second chart in Figure 4.16 outlines the results obtained for the Xilinx chip. Note that is no relationship between the maximum clock frequency and the fitness length on the Xilinx chip. The clock frequency of the system generally varies between 30 and 25 MHz.

This difference in behavior between FPGAs occurs due to differences in the interconnect architecture and nature of the basic programmable units for each brand of FPGA. In Altera chips all communication between Logic Array Blocks uses the global FastTrack interconnect though there is some small connection between Logic Elements inside the Logic Array Blocks. Xilinx chips use direct wires, horizontal and vertical general-purpose wires and long lines for high fanout, low skew signals. Since all FastTrack wires are identical, interconnection delays are more predictable but harder to optimize for performance. In Xilinx FPGAs, interconnections may involve varying numbers of small and long wire segments making it difficult to predict interconnection delays. Therefore, during the placement phase of the synthesis process, there is more potential for performance optimizations of interconnect resources in the Xilinx chips. This optimization process reduces the affect of increasing cascaded circuitry used for the comparator in the Tournament Arbiter as well as other optimizations for components connected to the Fitness RAM multiplex bus. Altera Logic Elements are explicitly linked to one another via the carry or cascade circuitry inside, making it difficult for the CAD tool to successfully map the corresponding comparator into the device. Since all communication between components and the Fitness RAM multiplex bus are done using the FastTrack interconnect, the CAD tool will have less flexibility with respect to optimizing the length of the internal connections. Hence, there is a trade-off between decreasing predictable performance on the Altera chip as the length of the fitness

representation changes, versus variable performance within a range of 25-35 MHz on the Xilinx chip. Depending on the nature of the application using the evolvable hardware approach, one type of behavior may be preferred over the other.

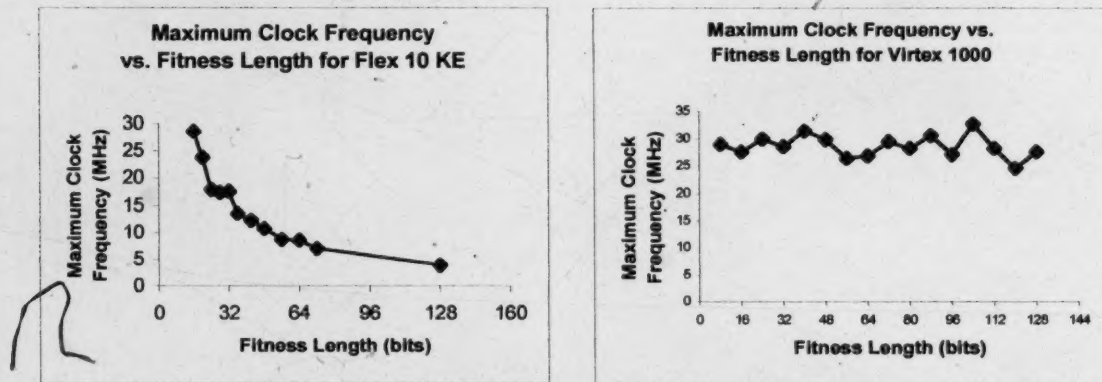


Figure 4.16 Effects of Fitness on System Clock Frequency
Chromosome Length: 8-bits, Population Size: 256 individuals

In both implementations, internal RAM is added to the system. In the Altera chip, the Embedded Array Blocks are separate from the Logic Array Blocks, but the clock frequency is still reduced from 37 MHz to 29 MHz. In the Xilinx chip, internal memory is implemented using Block SelectRam; the clock frequency of the system is reduced from 48 MHz to between 25-30 MHz. These observations occur when the fitness representation is 16-bits, chromosome length is 8-bits and population size is 256 individuals

4.6 Bottlenecks

By analyzing the maximum clock frequency and execution times of each component in the Genetic Algorithm, it is apparent that sources of overhead come from the Controller, Tournament Selection, Reproduction and Population Replacement. Speeding up these three processes can reduce the execution time for one generation significantly. Generally the Evaluation Component is the most time consuming portion of a Genetic Algorithm's execution time. However, in this simple application a simple steady-state pipeline is used to speed this process up.

4.6.1 Termination Criteria

The termination criterion is determined by incrementing the number of generations each time the optimization process completes a cycle until a maximum number of generations are reached. This value is provided externally however, if the maximum limit is $2^{20} - 1$ or 1,048,575, resulting in

the use of a 20-bit counter inside the Controller component, the Controller operates at a maximum clock frequency of 25.64MHz. When the size of this counter was reduced to $2^{10} - 1$ or 1023, the maximum clock frequency of the system increased to 43.47MHz in the Altera chip. Likewise, the clock frequency increased from 73.56 MHz to 90.58 MHz when the counter length was reduced from 20 bits to 10 bits in the Xilinx chip.

4.6.2 Tournament Selection

The Selection process is the second longest process in the Genetic Algorithm. Clocked at 20MHz, it takes 121.7 μ sec to complete. In Tournament Selection the Candidate Selector and Tournament Arbiter take turns using resources. Approximately one half of the execution time of the Selection process could be reduced if the Candidate Selector can continue obtaining new random candidates while the Tournament Arbiter is determining which Candidate is written to the Intermediate Population. This way one component is not sitting idle while the other is activated. This would require some new handshaking signal modifications but would not be an arduous task.

4.6.3 Reproduction

The Reproduction process is the longest process in the Genetic Algorithm, taking 180.45 μ sec, clocked at 20MHz. As in Tournament Selection, the Probability Selector and Reproduction components take turns using resources while the other sits idle. The Probability Selector could continue obtaining new random probabilities and parents while the Reproduction Component works on a different set of parents. The Reproduction Component could be broken down even further, crossover on one set of parents can occur while mutation and inversion occur on a different set of offspring. By dividing the tasks up into components that run and communicate with one another simultaneously, the runtime of the reproduction process should be decreased significantly. Again, some new handshaking signals would need to be implemented but it would not change the overall system design significantly.

4.6.4 Population Replacement

The Population Replacement component can be modified easily to be faster too. Clocked at 20MHz, its runtime is 51.3 μ sec. In this implementation, reading data from the Intermediate Population RAM takes two clock cycles for each individual. The Intermediate Population RAM sits idle while the new chromosome is written back to the Population RAM. This Population Replacement component could be modified such that reads from Intermediate Population RAM are occurring while the writes to Population RAM occur.

4.7 Conclusions

Thus the first step in creating an Adaptive Hardware System has been achieved. Synthesis of a flexible and scalable hardware-based Evolutionary Computation System is plausible on Field Programmable Gate Arrays. Separating the Genetic Algorithm from the Evaluation Component can increase the adaptability of the hardware-based system to many types of numerical optimization problems. FPGA manufacturers had laid a solid foundation for the AGM to be synthesized on any chip. Due to the varying internal design architectures of FPGAs, chips from different manufacturers have different attributes that can be desirable depending on the application. Xilinx chips yield the best system performance though it is not predictable. Chips from Altera are more predictable in performance due to the FastTrack Interconnects, however, the system performance is more limited. Antifuse technology used in Actel chips allows the chip to be non-volatile, meaning an extra ROM component is not needed to hold the program when the chip is powered up. Actel chips are one time programmable, which may or may not be desirable depending on the application. When synthesizing the Autonomous Genetic Machine on a FPGA the chromosome length will affect the maximum allowable population size on the chip. Depending on the FPGA fitness representation can have an effect on the system performance too. Increasing the population size has a minimal effect on system clock frequency. The execution time of the AGM alone is about 470 μ s when clocked at 20 MHz using a population of 256 individuals. If a very efficient, pipelined External Evaluation Component is used with the Autonomous Genetic Machine, approximately 2,500 generations can be executed in 1 second if the system is clocked at 20MHz.

Chapter Five: Fitness Evaluation Subsystem

Our objective is to develop an extension to the Autonomous Genetic Machine called the Fitness Evaluation Subsystem (FES). The Adaptive Hardware System discussed in this section utilizes the autonomous Genetic Algorithm engine, Autonomous Genetic Machine (AGM), and a generic training wrapper, Fitness Evaluation Subsystem (FES), capable of configuring the application using the chromosomes supplied from the AGM. The Fitness Evaluation Subsystem cycles through input training data, passes it through the application under test, and assesses the application responses to produce a corresponding fitness representation. Performance measurements and design considerations for the FES are investigated. Simulations utilizing the AGM together with the FES are analyzed in some simple linear regression problems.

5.1 Adaptive Hardware System Overview

The Adaptive Hardware System (AHS) is broken into three main components. The first component consists of the abstracted portions of the Genetic Algorithm that can run autonomously, independent of the external environment. It performs initialization, selection, reproduction, population replacement and checks on the termination criteria. The second component is responsible for training a configurable application by applying the chromosome to the configurable application, cycling through training input data and comparing the application responses to target responses in order to characterize the application's phenotype as fitness value. The third component is the application, capable of reconfiguring itself when it receives a new chromosome configuration. Figure 5.1, illustrates the concept. The Autonomous Genetic Machine accepts the threshold values for crossover and mutation, the maximum number of generations and the seed value for the random number generator. It begins the evolutionary process when it receives a Start signal. The AGM stores and manipulates the population and associated fitness representation internally in RAM. When it is time to evaluate the population, each chromosome is sent to the Fitness Evaluation Subsystem (FES). The FES cycles through the training data and reads the application responses comparing them to the target responses. FES stores this data in two internal ROM components. The application responses for a particular chromosome are characterized as fitness value that is sent back to the AGM.

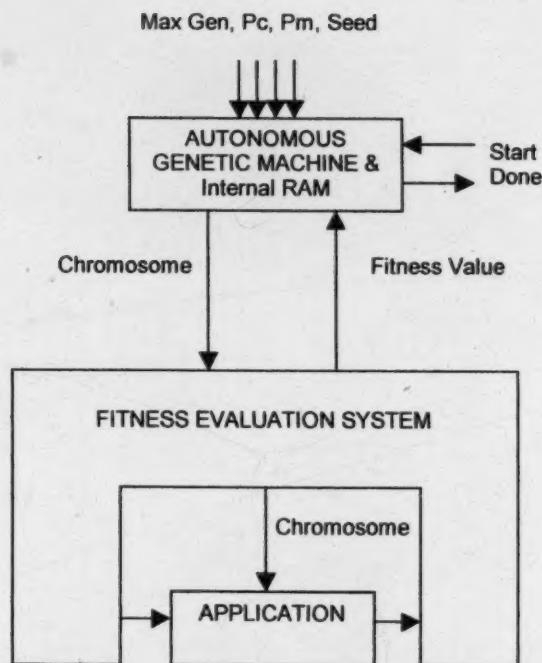


Figure 5.1 Concept of Adaptive Hardware System

5.2 Design of Fitness Evaluation Subsystem

The additional hardware overhead required by the Fitness Evaluation Subsystem is minimal. The Fitness Evaluation Subsystem interfaces with the Autonomous Genetic Machine in exactly the same manner as the polynomial functions presented earlier. The AGM sends the chromosome along with a data valid signal and receives the fitness representation along with another data valid signal from the FES. The FES contains a simple controller to synchronize the evaluation process, and an Input Preparation Unit and Output Processing Unit that interfaces to the application. As a consequence of this design architecture, the FES enables the AGM to be used in a broad range of optimization problems.

5.2.1 High Level System Architecture

The Fitness Evaluation Subsystem is divided into three major components synchronized by one controller. The four major subsystems are:

1. Input Preparation Unit
 - Retrieves training input from memory and sends the data to the application.
2. Output Processing Unit
 - Receives the response from the application and compares it to the target responses
 - Characterizes the number of correct responses as a fitness value representation
3. Training Controller
 - Passes the chromosome on to the application
 - Coordinates the Input Preparation Unit and Output Processing Unit during the process of calculating the fitness value
 - Sends the fitness value back to the AGM
4. Application Dependant System
 - Maps the chromosome to the configurable/evolvable application.
 - Processes input training data and passes output system responses to the Output Processing Unit for further analysis.

Internal read-only memory is used to store data for assessing application responses since the process of calculating a fitness value for a given chromosome requires the use of input training and target data. This internal ROM is configured prior to synthesis onto the FPGA. There are two memory components:

Training Memory: Input Data – stores the training input data for the application for use by the Input Preparation Unit

Target Memory: Output Data – stores the target responses for the application for used by the Output Processing Unit

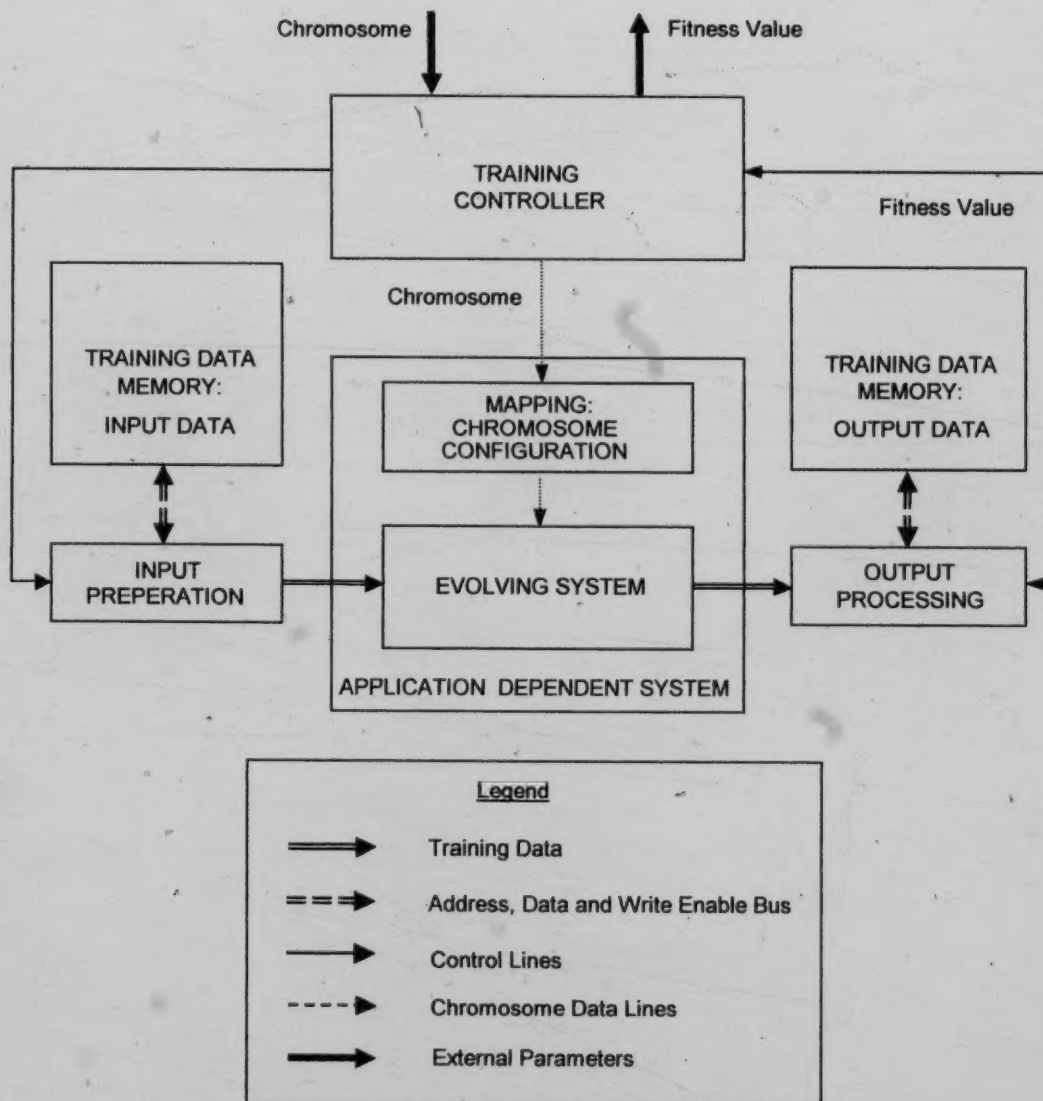


Figure 5.2 Architecture of Fitness Evaluation Subsystem

5.2.2 Training Controller

The Training Controller essentially coordinates the evaluation process in order to characterize a chromosome with a fitness value. When it receives a valid chromosome, it passes the chromosome to the application and activates the Input Preparation Unit. The Input Preparation Unit signals back that input is ready to be sent to the application. Then, the Training Controller activates the Output Processing Unit. The Output Processing Unit assesses the application response and signals back to the Training Controller that it is ready for another response. Calculations for the fitness value are done in the Output Processing Unit. The Training Controller synchronizes these processes until all the training data has been cycled through. Finally, a valid fitness value is sent from the Output Processing Unit to the Training Controller. The Training Controller passes the fitness value back to the AGM. The Training Controller synchronizes Fitness Evaluation Subsystem processes as follows:

STATE CLEAR:

- Clear the chromosome and fitness value registers
- NEXT STATE is GET CHROMOSOME

STATE GET CHROMOSOME:

- If CHROMOSOME_VALID signal received
 - Store Chromosome in register
 - NEXT STATE is INPUT PREPARATION ENABLE
- Else
 - Clear Chromosome in register
 - NEXT STATE is GET CHROMOSOME

STATE INPUT PREPARATION ENABLE:

- Send REQUEST signal to Pattern Generator
- NEXT STATE is OUTPUT PROCESSING ENABLE

STATE OUTPUT PROCESSING ENABLE:

- Send REQUEST signal to Response Analyzer
- NEXT STATE is OUTPUT PROCESSING WAIT

STATE OUTPUT PROCESSING WAIT:

- If OUTPUT_ASSESSED signal received
 - NEXT STATE is GET FITNESS
- Else
 - NEXT STATE is OUTPUT PROCESSING WAIT

STATE GET FITNESS:

- If OUTPUT_PROCESSING_DONE signal received
 - Store Fitness Value from Response Analyzer in register
 - Send Fitness Valid signal out of FES along with Fitness Value
 - NEXT STATE is DONE
- Else
 - Clear fitness value register
 - NEXT STATE is INPUT PREPARATION ENABLE

STATE DONE:

- NEXT STATE is CLEAR

5.2.3 Input Preparation Unit

The Input Preparation Unit reads the training input data from ROM and passes it to the Application-Dependant System. It contains an internal address counter to access ROM. Each time the Input Preparation Unit is activated the counter is incremented by one. When the address counter reaches the last memory value in memory, the Input Preparation Unit signals that it has completed cycling through the training input data to the Evaluation Controller.

5.2.4 Output Processing Unit

The Output Processing Unit receives the responses from the application and compares it to the associated target response. It too contains an internal address counter to access ROM containing the target response values. When it is activated the counter is incremented by one, the target response is obtained from memory and compared to the application response. The fitness for a given individual is calculated using the raw fitness, as follows,

$$r(i,t) = \sum | S(i,j) - C(j) |,$$

where $r(i,t)$ is the raw fitness of an individual i , at time, t , $S(i,j)$ is the fitness value returned by the individual for case j , $C(j)$ is the correct or target value for the fitness case j . The raw fitness is simply a measure of error. The lower the value the better the more accurate the results [11]. After the Output Processing Unit compares the application response with the target response and calculates the sum of error, it signals that it has finished processing the application response. When the Output Processing Unit has cycled through all the target responses, it signals that it has completed operation and the fitness value is passed back to the Evaluation Controller.

5.2.5 Application-Dependant System

The Application-Dependant System essentially consists of a mapping component that translates the chromosome into a configuration to be used by the evolving system. The focus for designing such systems switches to design for configurability. Digital applications that fit this method will need to be designed such that they can be reconfigured through binary valued chromosomes. Due to the strictness of the digital domain, such a configurable application may need to have asynchronous and synchronous portions that can be manipulated by the evolutionary process.

5.3 Linear Regression

In such an adaptive approach to digital system design the focus switches to design for configuration. Applications designed for the Adaptive Hardware System would have to be created such that simply applying a configuration bit vector can reconfigure them. This configuration bit vector would have to be designed to be compatible with the evolutionary approach where the configuration bit vector directly translates to a chromosome.

In this experiment, the application consists of a digital circuit modeling the equation $f(x) = Ax + B$. The coefficients A, and B are each represented as an 8-bit signed bit vector yielding a chromosome length of 16-bits. The Fitness Evaluation Subsystem cycles contains the input and target data describing a simple linear relationship between $f(x)$ and x . The goal of the AGM is to find the coefficients for the equation that best describes the data set with the minimum amount of error. Six data sets are constructed in order to test the effectiveness of the AGM approach. The first two data sets are based on the functions $f(x) = 25x - 11$ and $f(x) = -7x + 117$. The goal is two determine whether or not the AGM is capable of finding a solution close to or exactly matching the target functions when the slope is positive and negative. In the next four tests, noise is added to the data sets in order to assess how the AGM behaves. In each subsequent test, more noise is added. Each test was run for 100 generations using 100% crossover and 30% mutation clocked at 20MHz with a population size of 32 and a data set with 16 points. Table 5.1 and Figure 5.3 summarize the results.

Target Function	Result	Error	Average Error per Data Point	Seed	Execution Time (μ sec)
$f(x) = 25x - 11$	$f(x) = 24x - 3$	57	4.0000	0xEC06	52.7798
$f(x) = -7x + 117$	$f(x) = -5x + 99$	112	8.0000	0xD1A4	52.7835
Noise #1	$f(x) = 25x - 15$	148	9.2500	0x71E3	52.7825
Noise #2	$f(x) = 6x + 15$	329	21.5000	0x183D	52.7803
Noise #3	$f(x) = 4x + 128$	258	16.3750	0x1395	52.7879
Noise #4	$f(x) = 3x + 112$	460	30.6875	0xE16D	52.7826

Table 5.1 Experimental Results For Linear Regression Tests
Chromosome Length: 16-bits, Fitness Representation: 16-bits, Population Size: 32,
100% crossover, 30% mutation, 100 generations

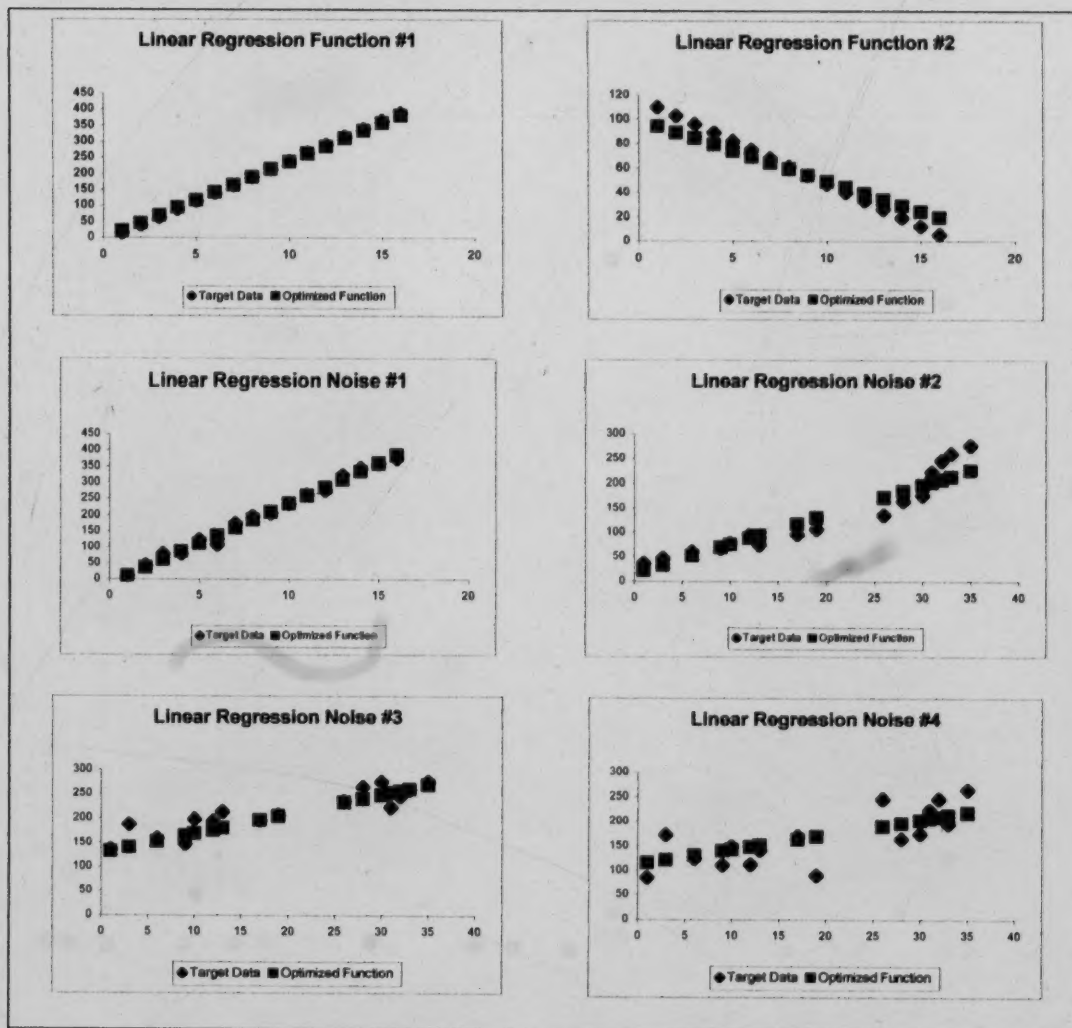


Figure 5.3 Comparison of Target Data with Optimized Results from AGM

Results indicate that the AGM is able to find reasonable solutions using linear approximation for data sets with and without noise. Hence, it has been shown that a more sophisticated problem, involving input training data and target data, can be addressed by adding the Fitness Evaluation Subsystem to the Autonomous Genetic Machine. In this experiment, the AGM and FES are used successfully in simple linear regression optimization problems.

5.4 Analysis of Fitness Evaluation Subsystem

Design considerations for synthesis of the Fitness Evaluation Subsystem on the Flex 10 KE FPGA from Altera and Virtex 1000 FPGA from Xilinx are now assessed. The affects of fitness representation on both chips are investigated.

5.4.1 Synthesis of FES

Table 5.4 and 5.5 summarize the number of basic configurable elements, and maximum clock frequency used by each sub-component in the Fitness Evaluation Subsystem on the Flex 10 KE and Virtex 1000 FPGAs. The implementation of the FES results in a small hardware overhead to the AGM; only approximately 430 extra logic cells are needed on the Altera chip and 126 extra slices on the Virtex chip. The Output Processing Unit limits the clock frequency of FES since calculations of the standardized fitness involves the use of addition, subtraction and comparison operations. The clock frequency of the Output Processing Unit will then mainly depend on the length of the associated fitness representation and the length of the data output from the configurable digital circuit.

Component	Logic Cells	Clock Frequency (MHz)
Training Controller	83	96.15
Input Preparation	75	85.47
Output Processing	321	30.39
FES	431	27.7

Table 5.4 Logic Cell Use and Clock Frequency of FES on Flex 10KE FPGA

Component	Slices	Clock Frequency (MHz)
Training Controller	32	97.49
Input Preparation	28	111.18
Output Processing	80	42.56
FES	126	45.18

Table 5.5 Logic Cell Use and Clock Frequency of FES on Virtex 1000 FPGA

5.4.2 Effects of Fitness Representation

Since the Output Processing unit uses addition, subtraction and comparison operations to calculate the standardized fitness of an individual, the length of representation of the fitness value can affect the system performance significantly. As in the Autonomous Genetic Machine, Figure 5.4 illustrates how varying the length of the fitness representation, will affect the system performance of the Altera and Xilinx chip.

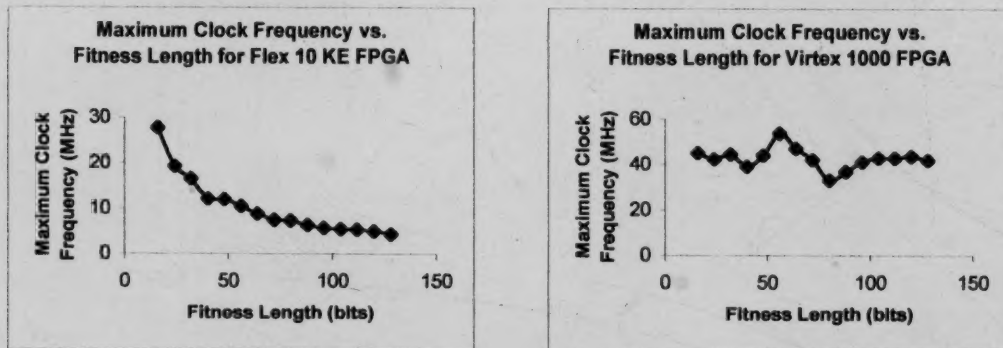


Figure 5.4 Effects of Fitness Representation on Maximum Clock Frequency for Flex 10 KE and Virtex FPGA

The FES exhibits similar behavior on the Altera and Xilinx chip to the AGM with regards to the Fitness Representation. As the Fitness Representation increases in length the clock frequency of the system decreases inversely on the Altera chip due to the FastTrack Interconnect internal design architecture. However, the Xilinx chip shows no relationship since its internal architecture consists of long, short interconnections between Configurable Logic Blocks. As in the AGM, the Altera chip offers predictability with a sacrifice in system performance as the fitness representation changes, whereas the Xilinx chip is less predictable offering reliable system performance within 33-53 MHz.

5.5 Conclusions

The Fitness Evaluation Subsystem has been successfully integrated with the Autonomous Genetic Machine. Capable of evaluating and cycling through input and target data of each chromosome, the Fitness Evaluation Subsystem increases the scope of potential problems that can be optimized with the Autonomous Genetic Machine. The Autonomous Genetic Machine and Fitness Evaluation Subsystem are designed to be generic enough to allow portability to any application provided the interface requirements are met. Simple linear regression optimization problems can be solved using this approach; in simulations, each test executes in approximately 52 μ sec over 100 generations given a population size of 32 using a data set with 16 points.

The Fitness Evaluation Subsystem together can be synthesized on any FPGA utilizing few resources. The synthesis of the FES on the Altera chip offers predictability with a sacrifice in system performance as the fitness representation changes, whereas the Xilinx chip is less predictable offering reliable system performance within a clock frequency range. The flexibility offered in the design of the Fitness Evaluation Subsystem and Autonomous Genetic Machine could be easily extended to other System-On-A-Chip Evolvable Hardware applications.

Chapter Six: Hardware Implementation

The Autonomous Genetic Machine and Fitness Evaluation Subsystem are synthesized and developed on the XSV 800 Board from XESS Corporation using a Virtex 800 FPGA from Xilinx. A brief description of the high-level design, execution steps and pin assignments are given. Linear regression problems are executed and compared with simulation results.

6.1 High Level Design

The top level of the chip consists of the Autonomous Genetic Machine, Fitness Evaluation Subsystem and Status Level Indicators. External RAM on the XSV Board is used instead of internal RAM making the system easier to test and debug. Two RAM banks exist on the board each containing 512K addresses of 16-bit data. Since external SRAM chips are used to store the population and fitness values, a new interface is added to communicate between the SRAM chips and Adaptive Hardware System components. Read and write operations take two clock cycles as opposed to the internal RAM components used in simulations that perform read and write operations in two and one clock cycles respectively. Though this increased waiting period does slow the execution time for one generation, it is not noticeable when executed on the XSV Board. Pc, Pm, Pi, Random Seed and Max Gen are specified in the main VHDL file for the Autonomous Genetic Machine before synthesis of the hardware system. The Autonomous Genetic Machine uses the left SRAM bank on the XSV Board. The main population chromosomes are stored between addresses 0x000 to 0x0FF as 16-bit vectors. Intermediate population follows from addresses 0x100 to 0x1FF. Fitness Values are stored between addresses 0x200 and 0x2FF as 16-bit unsigned integers. Following the complete execution of the Genetic Algorithm, the Intermediate Population and main Population should be exactly identical and serves as way of verifying correct results. As stated earlier, the Autonomous Genetic Machine communicates with the Fitness Evaluation Subsystem by sending the chromosomes along with a data valid signal externally and receiving the fitness representation along with another data valid signal. The Fitness Evaluation Subsystem uses the right SRAM bank on the XSV Board. Output Target Data resides in memory locations 0x00 to 0x0F and Input Training Data resides in memory locations 0x10 to 0x1F. The Fitness Evaluation Subsystem contains the linear regression application internally. The Status Level Indicators convey the current execution status of the genetic algorithm to the bar graph display on the XSV Board. The genetic algorithm is run for 1000 generations. When 250 generations pass the first LED lights. After 500 generations the second LED lights. After 750 generations pass the third LED lights. Finally, when the genetic algorithm has finished execution to last LED lights. These level indicators can be programmed for any number of generations by updating the definitions in the main VHDL package file. The top level of the design places the components on the FPGA as follows:

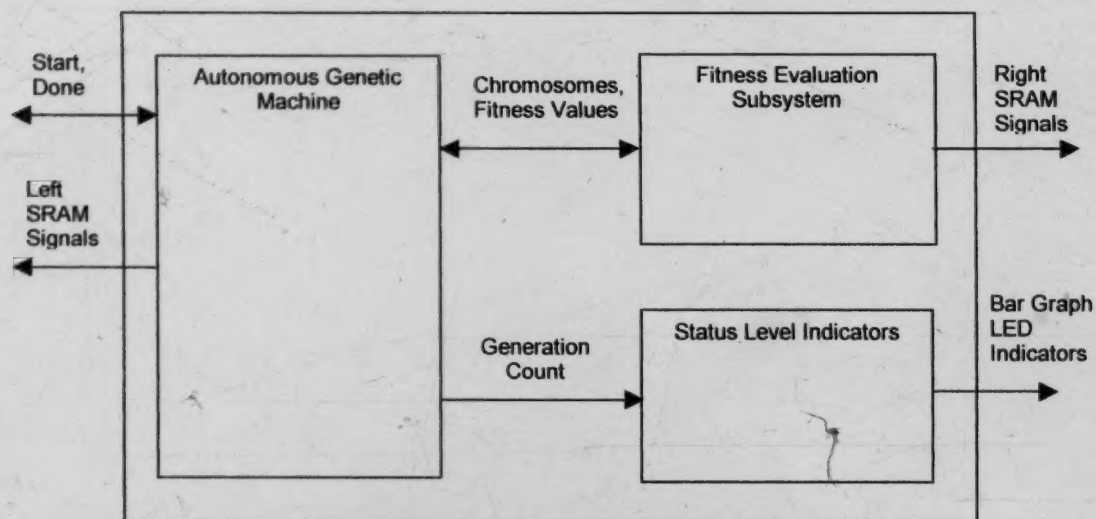


Figure 6.1 Top Level of Design on FPGA

6.2 Execution Steps and Pin Assignments

This Adaptive Hardware system is run in conjunction with a PC-to-SRAM interface. Since the contents of the SRAM do not change when the FPGA is reconfigured, the Adaptive Hardware System is run first, followed by the PC-to-SRAM interface in order to read the contents of the SRAM back to the PC. The execution steps are as follows:

1. Download the right PC-to-SRAM interface to the FPGA,
2. Write the training data binary file to the right SRAM bank using the XSV SRAM Utility software,
3. Download the left PC-to-SRAM interface to the FPGA,
4. Write logic '1' to all memory locations in the left SRAM bank using the XSV SRAM Utility software,
5. Download and run the Adaptive Hardware System to the FPGA and wait for DONE signal,
6. Re-download the PC-to-SRAM interface to the FPGA and read the left SRAM image back to the PC as a binary file.

Currently the system is clocked at 25 MHz. Pin assignments used in synthesis of this Adaptive Hardware System are given below. Lower case 'n' implies active-low logic.

Port Name	Direction	Description	Pin Number
Clock	Input	System Clock	89
Resetn	Input	Push-button 1 System Reset	174
GA_Startn	Input	Push-button 4 Start Signal	185
GA_Done	Output	Bar Graph System Done Signal	171
Level 0	Output	Bar Graph Operation Status Indicator	169
Level 1	Output	Bar Graph Operation Status Indicator	168
Level 2	Output	Bar Graph Operation Status Indicator	173
Level 3	Output	Bar Graph Operation Status Indicator	131
lAddr(18:0)	Output	Connects to the left SRAM bank address lines	229,230,231,232,234,235,236,237,238,187,188,189,191,192,193,194,195,199,200
lData(15:0)	Bi-dir	Connects to the left SRAM bank data lines	224,223,222,221,220,218,217,216,215,209,208,207,206,205,203,202
lcen	Output	Connects to the left SRAM /CE (chip enable) pin	186
loen	Output	Connects to the left SRAM /OE (output enable) pin.	228
lwen	Output	Connects to the left SRAM /WE (write enable) pin	201
rAddr(18:0)	Output	Connects to the right SRAM bank address lines	96,97,99,100,101,102,103,107,108,53,54,55,56,57,63,64,65,66,67
rData(15:0)	Bi-dir	Connects to the right SRAM bank data lines	94,93,87,86,85,84,82,81,80,79,78,74,73,72,71,70
rcen	Output	Connects to the right SRAM /CE (chip enable) pin	109
roen	Output	Connects to the right SRAM /OE (output enable) pin	95
rwen	Output	Connects to the right SRAM /WE (write enable) pin	68

Table 6.1 Pin Assignments of Adaptive Hardware System on XSV Board

6.3 Experimental Results Using Linear Regression

The Adaptive Hardware System is tested using linear regression problems, and results are compared with those from simulations. Table 6.2 summarizes results obtained from simulations of the Autonomous Genetic Machine and Fitness Evaluation Subsystem using six different linear regression problems. The first two tests involve data sets are defined explicitly by a linear function. The remaining data sets integrate noise of varying degrees. Table 6.3 summarizes results obtained from the actual hardware implementation of the Autonomous Genetic Machine and Fitness Evaluation Subsystem on the XSV 800 Board. Note that when different seed values are used, the implemented Adaptive Hardware System still obtains results close to the simulated results.

Target Function	Result	Error	Average Error per Data Point	Seed
$f(x) = 25x - 11$	$f(x) = 24x - 3$	57	4.0000	0xEC06
$f(x) = -7x + 117$	$f(x) = -5x + 99$	112	8.0000	0xD1A4
Noise #1	$f(x) = 25x - 15$	148	9.2500	0x71E3
Noise #2	$f(x) = 6x + 15$	329	21.5000	0x183D
Noise #3	$f(x) = 4x + 128$	258	16.3750	0x1395
Noise #4	$f(x) = 3x + 112$	460	30.6875	0xE16D

Table 6.2 Simulations Results For Linear Regression Tests
Chromosome Length: 16-bits, Fitness Representation: 16-bits, Population Size: 32,
100% crossover, 30% mutation, 100 generations

Target Function	Result	Error	Average Error per Data Point	Seed
$f(x) = 25x - 11$	$f(x) = 24x$	70	4.3750	0xE596
$f(x) = -7x + 117$	$f(x) = -9x + 127$	152	9.5000	0x18E3
Noise #1	$f(x) = 24x - 1$	148	9.2500	0xA4C8
Noise #2	$f(x) = 5x + 29$	339	21.1875	0x27C9
Noise #3	$f(x) = 5x + 113$	337	21.0625	0x39AF
Noise #4	$f(x) = 4x + 90$	462	28.8750	0xB264

Table 6.3 Implementation Results For Linear Regression Tests
Chromosome Length: 16-bits, Fitness Representation: 16-bits, Population Size: 256,
100% crossover, 30% mutation, 1000 generations

6.4 Conclusions

The Autonomous Genetic Machine and Fitness Evaluation Subsystem can be executed in a real hardware system. The XSV 800 Board from XESS Corporation using a Virtex 800 FPGA provides a suitable platform on which this Adaptive Hardware System can be implemented. When tested with linear regression problems, results from simulations are similar to those obtained from the actual hardware implementation.

Chapter Seven: Conclusions and Future Work

7.1 Conclusions

Development of evolvable hardware systems using Field Programmable Gate Arrays is possible and can be designed with a general-purpose architecture capable of being used for a variety of optimization problems. The Autonomous Genetic Machine (AGM) is the realization of such an architecture that can be ported to many different FPGA chips due to the flexibility offered by VHDL. The AGM uses modular components and well-defined re-useable interfaces giving it the advantage of being applied to a variety of hardware-based optimization problems. Adding another layer of functionality to the AGM capable of storing and cycling through training input and target data for the evaluation of each individual in the population increases the number of potential applications it can be used in. This Fitness Evaluation Subsystem (FES) acts as a training wrapper for the target application.

Simulations show that the AGM can solve numerical optimization problems of varying difficulty involving locating the optima in three-dimensional space. When synthesizing the Autonomous Genetic Machine on a FPGA the chromosome length will affect the maximum allowable population size on the chip. Depending on the FPGA fitness representation can have an effect on the system performance too. Increasing the population size has a minimal effect on system clock frequency. The execution time of the AGM alone is about 470 μ s when clocked at 20 MHz using a population of 256 individuals. If a very efficient Evaluation Component is used with the Autonomous Genetic Machine, approximately 2,500 generations can be executed in 1 second if the system is clocked at 20MHz.

FPGA manufacturers had laid a solid foundation for the AGM to be synthesized on any chip. Due to the varying internal design architectures of FPGAs, chips from different manufacturers have different attributes that can be desirable depending on the application. Xilinx chips yield the best system performance though it is not predictable. Chips from Altera are more predictable in performance due to the FastTrack Interconnects; however, the system performance is more limited. Antifuse technology used in Actel chips allows the chip to be non-volatile, meaning an extra ROM component is not needed to hold the program when the chip is powered up. Actel chips are one time programmable, which may or may not be desirable depending on the application.

Adding the Fitness Evaluation Subsystem to the Autonomous Genetic Machine, changes the nature of the optimization problems to which it can be applied. Simple linear regression optimization problems can be solved using this approach; in simulations, each test executes in

approximately 52 μ sec over 100 generations given a population size of 32 using a data set with 16 points. The flexibility offered in the design of the Fitness Evaluation Subsystem and Autonomous Genetic Machine could be easily extended to other System-On-A-Chip Evolvable Hardware applications, possibly giving hardware systems the capability of on-line adaptability.

Implementation of the Adaptive Hardware System is possible utilizing the XSV 800 Board from XESS Corporation. Experimental results show that linear regression experiments can be developed and executed successfully on the Virtex FPGA from Xilinx. Thus, the first step towards creating System-On-A-Chip Evolvable Hardware with on-line adaptability is achieved.

7.2 Future Work

The most important underlying component of the Autonomous Genetic Machine is the pseudo-random number generator implemented as a Linear Feedback Shift Register. The random number generator used in this work is cyclical, and based on using internal XOR gates or taps between D flip-flop registers containing an initial seed value. The taps are positioned inside the register such that the LFSR will sequence through every possible value before returning to the initial seed value without repetition. Though this is the most commonly used random number generating technique, another approach exists, called Cellular Automata. In this approach, D flip-flops are grouped together as cells that implement two kinds of functions, Rule 90: $s_{i+1} = s_{i-1} \oplus s_{i+1}$ and Rule 150: $s_{i+1} = s_{i-1} \oplus s_i \oplus s_{i+1}$. Cells are assembled such that Rule 90 and Rule 150 are alternated. Since this approach has proven to be superior to the shifting nature of the LFSR, an analysis of the effects of using different pseudo-random number generations techniques on the Autonomous Genetic Machine could shed some interesting results.

Since the Canonical Genetic Algorithm is implemented in this work as the Autonomous Genetic Machine, elitism is not used. In elitism, a subset of the best individuals from the previous generation are saved and allowed to propagate to the next generation. This ensures that the parent selection process does not accidentally overlook the fittest individuals. Elitism was not implemented due the significant amount of hardware overhead required. This would involve cycling through the entire population and selecting or sorting the population and saving the best individuals in the intermediate population. However, the effectiveness of elitism should not be overlooked simply due to increased hardware complexity. Investigations into improving the effectiveness of the Autonomous Genetic Machine could prove fruitful.

Due to the evaluation time required to calculate the fitness of every individual in the population, Genetic Algorithms are generally an intensive computational process. However, if a pipelined architecture is used to assess the fitness of every individual, then the rest of the functional components inside the AGM become the most time consuming tasks. Since the functional components run in sequence it is possible to implement parent candidate selection while the tournament between two other candidates are occurring at the same time. Also, reproduction can be broken into three distinct stages. In the first stage, the candidates are randomly paired to form parents. In the second stage, offspring are produced via crossover. Finally, mutations occur to the offspring. All three stages can be executed as a pipeline. The potential for speeding up execution of the AGM using a pipelined approach would also be a very practical extension of this work.

Steady-State Genetic Algorithms have been criticized for not evolving the population efficiently, though this approach ensures there are no copies of the same individual in the population. In the Steady-State approach, parents are only replaced if the offspring produced has a better fitness than the parents. This significantly reduces the amount of time taken to evaluate the population. Since the results from the AGM show that one generation can be evolved in less than 0.5 milliseconds using the Canonical approach, the slow evolution time can be ignored in the Steady-State approach. A comparison of the Canonical approach versus the Steady-State approach should be considered too before final implementation in FPGAs.

Investigations into different selection methods and reproduction operations could be focus of future work as well. Depending on the application one-point, two-point and uniform crossover may yield substantially different results. Likewise, a comparison of Roulette Wheel Selection and Stochastic Universal Selection versus Tournament Selection could prove to be interesting. The hardware based Genetic Algorithm presented in this study, implements selection as tournament selection since it is the simplest approach, using the least amount of hardware overhead and yielding the fastest execution time. However, depending on the focus of research and results, the added overhead and increased execution time using the other selection methods could be worth overlooking.

Still, other implementations using dedicated hardware are possible without using FPGAs. If a Genetic Algorithm could be written in assembly language and executed on a microprocessor, a comparison between the performance of the microprocessor and the FPGA could prove very useful. However, FPGAs offer a platform on which many kinds of design architectures can be developed. Exploitation of pipelining and parallelism is not possible on microprocessor that executes a single instruction at a time.

Once these issues have been firmly investigated, the next step in realizing a hardware system with on-line adaptive capabilities will be to develop the On-Line Monitoring and Diagnostic System in conjunction with an appropriate application.

Bibliography

1. Bäck, T. and Schwefel, H-P., "Evolutionary Computation: An Overview". Proceedings of the Third IEEE Conference on Evolutionary Computation 1996, pp. 20-29, IEEE Press, Piscataway NJ, 1996.
2. Banzhaf, W., Nordin, P., Keller, R.E., and Francone, F.D., *Genetic Programming: An Introduction On the Automatic Evolution of Computer Programs and Its Applications*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1998.
3. Brown, S.D., "Field-Programmable Devices - Technology, Applications, Tools, A Tutorial Overview" 2nd ed., University of Toronto course notes, 1996.
4. Coello, C. A., Christiansen, A. D., and Aguirre, A. H. *Use of Evolutionary Techniques to Automate the Design of Combinational Circuits*. International Journal of Smart Engineering System Design, 2000.
5. Goldberg, D.E., *Genetic Algorithms in Search Optimization and Machine Learning*, Addison-Wesley, 1989.
6. Haddow, P. C. and Tufte, G., "An Evolvable Hardware {FPGA} for Adaptive Hardware", *Proc of the 2000 Congress on Evolutionary Computation*, IEEE Service Center, Piscataway, NJ, pp. 553-560, 2000.
7. Higuchi, T., Iwata, M., Keymeulen, D., Sakanashi, H., Murakawa, M., Kajitani, I., Takahashi, E., Toda, K., Salami, M., Kajihara, N., and Otsu, N., "Real-World Applications of Analog and Digital Evolvable Hardware", *IEEE Transactions on Evolutionary Computation*, Vol.3, No.3, pp. 220-235, Sept. 1999.
8. Higuchi, T. et al. "A Gate-Level EHW Chip: Implementing GA Operations and Reconfigurable Hardware on a Single LSI", *Proc. of Int. Conf. on Evolvable Systems (ICES'98)* Springer-Verlag, Berlin, pp. 1-12, 1998.
9. Higuchi, T., Iwata, M., Kajitani, I., Iba, H., Hirao, Y., Furuya, T., and Manderick, B., *Evolvable Hardware and Its Applications to Pattern Recognition and Fault Tolerant Systems*, Springer-Verlag, Berlin, vol 1062, pp. 118- 135, 1996.
10. Holland, J.H., *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, 1975.
11. Koza, J.R., *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, Cambridge, MA, 1992.
12. Mazumder, P. and Rudnick, E.M., *Genetic Algorithms for VLSI Design, Layout & Test Automation*, Prentice Hall PTR, Upper Saddle River, NJ, 1999.
13. Miller, J.F., D. Job, and V.K. Vassilev, "Principles in the Evolutionary Design of Digital Circuits - Part 1", *Genetic Programming and Evolvable Machines*, Editor-in-Chief: Wolfgang Banzhaf, Kluwer Academic Publishers, Dordrecht, The Netherlands, Vol. 1, No. 1, pp.7-35, April 2000.
14. Poli, R. and Logan, B., "Evolutionary Computation Cookbook: Recipes for Designing New Algorithms". *Proceedings of the Second Online Workshop on Evolutionary Computation*, Nagoya, March 1996.

15. Scott, S. D., Seth, S., and Samal, A., *A hardware engine for genetic algorithms*. Technical Report UNL-CSE-97-001, University of Nebraska-Lincoln, June 1997.
16. Smith, D.J., *HDL Chip Design*, Doone Publications, Madison, AL, 1996.
17. Stoica, A., "Evolvable Hardware: From On-Chip Circuit Synthesis to Evolvable Space Systems.", *Proceedings of the 30th IEEE Symposium on multi-valued logic*, IEEE Press, Portland, USA, May 23-25, 2000.
18. Thompson, A., "Evolving fault tolerant systems", *Proc. 1st IEE/IEEE Int. Conf. on Genetic Algorithms in Engineering Systems: Innovations and Applications (GALESIA'95)*, IEE Conf. Publication No. 414, pp. 524-529, 1995.
19. Thompson, A., Layzell, P., and Zebulum, R., "Explorations in Design Space: Unconventional Electronics Design Through Artificial Evolution", *IEEE Transactions On Evolutionary Computation*, Vol.3, No.3, pp. 167-196, Sept. 1999.
20. Tommiska, M. and Vuori, J., "Implementation of genetic algorithms with programmable logic devices", *Proceedings of the Second Nordic Workshop on Genetic Algorithms and their Applications (2NWGA)*, J. T. Alander, Ed. pp. 71-78, August 1996.
21. Torresen, J., "A Scalable Approach to Evolvable Hardware". *Genetic Programming and Evolvable Machines*, Editor-in-Chief: Wolfgang Banzhaf, Kluwer Academic Publishers, Dordrecht, The Netherlands, Vol. 3, No. 3, pp.259-283, September 2002.
22. Torresen, J., "Reconfigurable Logic Applied for Designing Adaptive Hardware Systems". *International Conference on Advances in Infrastructure for Electronic Business, Education, Science, and Medicine on the Internet (SSGRR 2002W)*, L'Aquila, Italy January 2002.
23. Torresen, J., "Evolvable Hardware as a New Computer Architecture". *International Conference on Advances in Infrastructure for Electronic Business, Education, Science, and Medicine on the Internet (SSGRR 2002W)*, L'Aquila, Italy, January 2002.
24. Vassiliev, V.K. and Miller, J.F., "Scalability Problems of Digital Circuit Evolution: Evolvability and Efficient Designs", *Proceedings of the Second NASA / DoD Workshop on Evolvable Hardware*, IEEE Computer Society Press, Los Alamitos, CA, pp. 55-64, July 13 -15 2000.
25. Zebulum, R., Pacheco, M, and Vellasco, M., "Synthesis of CMOS Operational Amplifiers Through Genetic Algorithms", *Proceedings of the XI Brazilian Symposium on Integrated Circuit Design*, vol. 11, no. 1, pp. 125-128, 1998.
26. "Altera Devices". <http://www.altera.com/products/devices/dev-index.jsp>.
27. "Cognitive Systems and Evolutionary Computation at Battelle". <http://www.battelle.org/cogsys/evolcomp.htm>.
28. "FAQs for Spartan-II". http://www.xilinx.com/company/press/kits/spartan2/faq_sp2.htm.
29. "Virtex-E FAQ". http://www.xilinx.com/prs_rls/vtxefaq.htm.
30. "Welcome to Actel". <http://www.actel.com>.

31. "Xilinx Home Products and Solutions Silicon Solutions".
http://www.xilinx.com/xlnx/xil_prodcat_landingpage.jsp?title=Devices.
32. Xilinx Inc., "Xcell Journal: The Authoritative Journal For Programmable Logic Users".
Issues 41,42,43 Xilinx Inc, 2002.
33. Xilinx Application Note, "Understanding the CoolRunner-II Logic Engine".
http://www.xilinx.com/publications/products/cool2/apps_pdf/xapp376.pdf.